

C51 基础与应用实例

常喜茂 孔英会 付小宁 编著

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

本书通过多个典型应用实例详细地介绍了 C51 单片机各种应用设计, 首先介绍了 C51 单片机开发的基础知识, 然后通过多个非常具有实际应用价值的实例来介绍 C51 单片机各个模块的应用, 最后介绍 C51 单片机的几个典型的高级应用。

本书语言通俗、实例丰富、代码分析详尽, 有较强的实用性和参考价值, 适合大专院校计算机、电子、电气、控制及相关专业学生学习参考, 也可供单片机开发人员和系统设计人员参考使用。

本书源代码可从华信教育资源网(教育网: www.huaxin.edu.cn 或公共网: www.huaxin.com.cn) 免费注册后下载。

未经许可, 不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有, 侵权必究。

图书在版编目(CIP)数据

C51 基础与应用实例/常喜茂, 孔英会, 付小宁编著. —北京: 电子工业出版社, 2009.1

ISBN 978-7-121-08052-4

I. C… II. ①常…②孔…③付… III. 单片微型计算机 IV. TP368.1

中国版本图书馆 CIP 数据核字(2008)第 210385 号

责任编辑: 田宏峰

印 刷: 北京市李史山胶印厂
装 订:

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编: 100036

开 本: 787×1092 1/16 印张: 26.5 字数: 678 千字

印 次: 2009 年 1 月第 1 次印刷

印 数: 4 000 册 定价: 49.80 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线: (010) 88258888。

前 言

C51 单片机由于其出色的性价比, 以及具备简捷实用、系统完善的开发工具, 使它的应用遍及各个领域。

(1) 在智能仪表中的应用。单片机广泛应用于各种仪器仪表, 使仪器仪表实现智能化, 并提高了测量的自动化程度和精度, 简化了仪器仪表的硬件结构, 提高了其性价比。

(2) 在机电一体化中的应用。机电一体化是指集机械技术、微电子技术、计算机技术于一体, 具有智能化的特征, 这是机械工业发展的方向。单片机作为产品中的控制器, 发挥它的体积小、可靠性高、功能强等优点, 极大地提高了机器的自动化、智能化程度。

(3) 在实时控制中的应用。单片机广泛地用于各种实时控制系统中。例如, 利用单片机作为控制器, 在工业测控、航空航天、尖端武器、机器人等各种实时控制系统应用, 可使系统保持在良好的工作状态, 并提高系统的工作效率和产品质量。

(4) 在分布式多机系统中的应用。在复杂的系统中, 通常采用分布式多机系统。多机系统通常由若干台功能各异的单片机组成, 它们通过串行通信相互联系、协调工作, 并完成各自特定的任务。

(5) 在人们生活中的应用。在人们的日常生活中, 洗衣机、电冰箱、电子玩具、收录机等家用电器都应用了单片机, 提高了智能化程度, 增加了功能, 使人们的生活更加方便和舒适, 得到了人们的广泛接受。

从上述五个方面可以看出, 单片机的应用正在从根本上改变着传统的控制系统设计思想和设计方法, 它已经替代了以前很多必须由模拟电路或数字电路实现的控制。随着单片机应用技术的推广普及, 微控制技术必将不断发展, 日益完善, 更加充实。

本书是一本基础加实例的图书, 各章的内容包括: 第 1 章C51 单片机基础, 重点介绍C51 单片机的硬件基础知识; 第 2 章Keil 8051 C编译器, 主要介绍C51 的集成开发环境 μ Vision3; 第 3 章RTX51 实时操作系统, 主要介绍RTX51 实时操作系统。第 4 章常用的单片机芯片介绍, 主要介绍常用的 8 位单片机芯片; 第 5 章键盘与显示实例, 介绍几种典型的键盘与显示实例设计; 第 6 章C51 单片机控制实例, 主要介绍C51 单片机的一些典型的控制应用实例; 第 7 章数据采集系统实例, 主要介绍C51 单片机数据采集的应用实例; 第 8 章通信实例, 介绍几种典型的单片机通信实例; 第 9 章综合应用实例, 选择了一些C51 单片机典型的综合应用实例, 包括I²C、GPS、USB、以太网等, 这些实例具有很高的实际应用价值。本书源代码可从华信教育资源网(教育网: www.huaxin.edu.cn 或公共网: www.huaxin.com.cn) 免费注册后下载。

本书主要由常喜茂、孔英会、付小宁编著, 参加编写的人员还有姜艳波、兰婵丽、赵光、王波波、刘文涛、刘群、赵辉、吴丽、王烁、宋盟、丁玲、王丽娟、胡桂桃、姚国玲、王维晶、赵光, 在此表示感谢!

目 录

第 1 章 C51 单片机基础	(1)
1.1 C51 单片机基本介绍	(1)
1.1.1 引脚功能说明	(2)
1.1.2 C51 单片机的特点	(4)
1.2 C51 单片机的内部结构	(5)
1.2.1 CPU	(5)
1.2.2 存储器结构	(12)
1.2.3 片内并行接口	(18)
1.3 C51 单片机定时/计数器	(20)
1.3.1 定时/计数器结构	(20)
1.3.2 定时/计数器的方式控制字	(21)
1.3.3 定时/计数器工作方式	(22)
1.4 单片机的工作方式	(24)
1.4.1 单片机的复位方式	(25)
1.4.2 程序执行方式	(26)
1.4.3 节电工作方式	(27)
1.4.4 EPROM 编程和校验方式	(29)
1.5 C51 单片机的指令系统	(30)
1.5.1 计算机语言	(31)
1.5.2 C51 单片机的寻址方式	(32)
1.5.3 C51 单片机的指令系统	(38)
1.5.4 指令系统中的符号说明	(39)
第 2 章 Keil 8051 C 编译器	(54)
2.1 系统概述	(54)
2.2 使用 Keil 开发	(56)
2.2.1 μ Vision3 项目管理窗口简介	(56)
2.2.2 Keil C51 开发过程	(60)
2.2.3 Keil 的调试	(66)
2.3 汇编语言与 C 语言的混合使用	(73)
2.3.1 汇编语言与 C 语言的比较	(73)
2.3.2 C 语言中嵌入汇编语言	(76)
2.3.3 汇编语言程序调用 C 语言程序	(79)
第 3 章 RTX51 实时操作系统	(82)
3.1 RTX51 操作系统简介	(82)
3.1.1 实时操作系统 (RTOS)	(82)

3.1.2	RTX51 实时操作系统	(82)
3.2	软硬件需求与定义	(89)
3.3	RTX51 的功能函数	(92)
3.3.1	信号控制函数	(93)
3.3.2	任务控制函数	(95)
3.3.3	延时控制函数	(96)
3.4	建立 RTX51 Tiny 应用程序	(98)
第 4 章	常用的单片机芯片介绍	(100)
4.1	HOLTEK 公司 HT48XX 系列单片机介绍	(100)
4.1.1	HT48R05A-1	(100)
4.1.2	HT48R50A-1	(101)
4.1.3	HT48C50-1	(103)
4.2	Motorola 公司的 MC68HC08 系列单片机	(104)
4.2.1	MC68HC08AS32CFN	(104)
4.2.2	MC68HC08AS32FU	(105)
4.3	Philips 公司推出的改进型 C51 单片机	(107)
4.3.1	产品性能	(107)
4.3.2	内部框图及引脚说明	(108)
4.4	Atmel 公司的 AT89S 系列单片机	(110)
4.4.1	AT89S 系列单片机的特点	(110)
4.4.2	AT89S 系列单片机的引脚图及内部结构框图	(111)
第 5 章	键盘与显示实例	(113)
5.1	七段数码管显示	(113)
5.1.1	实例说明	(113)
5.1.2	七段数码管介绍	(113)
5.1.3	硬件电路设计	(114)
5.1.4	软件设计	(116)
5.2	单片机键盘程序 (4x4 矩阵式)	(118)
5.2.1	实例效果说明	(118)
5.2.2	硬件电路设计	(119)
5.2.3	软件程序设计	(119)
5.3	单片机控制 LCD 显示	(126)
5.3.1	实例说明	(126)
5.3.2	芯片介绍	(126)
5.3.3	硬件设计	(129)
5.3.4	软件设计	(129)
5.4	带有存储功能的数显温度计	(139)
5.4.1	实例说明	(140)
5.4.2	芯片介绍	(140)

5.4.3	硬件电路设计.....	(145)
5.4.4	软件设计	(146)
5.5	单片机实现数字电压表显示	(155)
5.5.1	实例说明	(155)
5.5.2	设计思路分析.....	(155)
5.5.3	硬件电路设计.....	(155)
5.5.4	软件设计	(156)
第 6 章	C51 单片机控制实例	(160)
6.1	基于 ISD4004 芯片的语音录放设计.....	(160)
6.1.1	实例说明	(160)
6.1.2	ISD4004 介绍	(160)
6.1.3	硬件设计	(168)
6.1.4	软件设计	(168)
6.2	单片机密码锁设计	(174)
6.2.1	实例说明	(174)
6.2.2	设计思路分析.....	(174)
6.2.3	硬件电路设计.....	(177)
6.2.4	软件设计	(179)
6.3	利用单片机 P1 口控制直流电动机.....	(192)
6.3.1	实例效果说明.....	(192)
6.3.2	74HC244 介绍	(192)
6.3.3	直流电动机	(194)
6.3.4	硬件设计	(194)
6.3.5	软件设计	(195)
6.4	单片机实现智能充电器的设计.....	(197)
6.4.1	实例说明	(197)
6.4.2	设计思路分析.....	(198)
6.4.3	芯片介绍	(199)
6.4.4	硬件电路设计.....	(204)
6.4.5	软件设计	(206)
6.5	基于 DS12C887 芯片的实时时钟日历显示.....	(209)
6.5.1	实例说明	(209)
6.5.2	DS12C887 芯片说明	(209)
6.5.3	硬件电路图设计.....	(213)
6.5.4	软件设计	(214)
6.6	单片机实现步进式 PWM 信号输出	(217)
6.6.1	实例说明	(217)
6.6.2	设计思路分析.....	(217)
6.6.3	硬件电路设计.....	(223)

6.6.4	软件设计	(224)
第 7 章	数据采集系统实例	(227)
7.1	基于 ADC0809 的并行 A/D 转换	(227)
7.1.1	实例说明	(227)
7.1.2	ADC0809 芯片介绍	(227)
7.1.3	硬件电路设计	(230)
7.1.4	软件设计	(231)
7.2	基于 TLC549 的串行 A/D 转换	(232)
7.2.1	实例说明	(233)
7.2.2	A/D 转换简介	(233)
7.2.3	TLC549 芯片介绍	(236)
7.2.4	硬件电路设计	(238)
7.2.5	软件设计	(239)
7.3	基于 MAX532 的串行 D/A 转换	(241)
7.3.1	实例说明	(241)
7.3.2	D/A 转换	(241)
7.3.3	MAX532 芯片介绍	(242)
7.3.4	硬件原理图的设计	(244)
7.3.5	程序设计	(245)
7.4	基于 DS18B20 的数字温度计设计	(248)
7.4.1	实例效果说明	(248)
7.4.2	DS18B20 芯片介绍	(248)
7.4.3	MAX7219 芯片介绍	(252)
7.4.4	硬件原理图的设计	(253)
7.4.5	软件设计	(255)
7.5	基于双口 RAM 的单片机间通信	(261)
7.5.1	实例分析	(261)
7.5.2	IDT7005 芯片介绍	(261)
7.5.3	硬件设计	(265)
7.5.4	软件设计	(267)
第 8 章	通信实例	(272)
8.1	单片机实现点对点的数据传输	(272)
8.1.1	实例说明	(272)
8.1.2	串行通信	(272)
8.1.3	MAX3232 芯片介绍	(274)
8.1.4	硬件原理图的设计	(275)
8.1.5	软件设计	(277)
8.2	单片机实现短距离无线通信	(283)
8.2.1	nRF401 介绍	(283)

8.2.2	PTR2000 的介绍	(287)
8.2.3	硬件设计	(288)
8.2.4	软件设计	(290)
第 9 章	综合应用实例	(297)
9.1	I ² C 总线接口技术在 IC 卡上的应用	(297)
9.1.1	实例说明	(297)
9.1.2	I ² C 接口技术	(297)
9.1.3	芯片 24LC01B 的介绍及应用	(299)
9.1.4	硬件设计	(300)
9.1.5	软件设计	(301)
9.2	C51 单片机实现 GPS 定位设计	(306)
9.2.1	实例效果说明	(306)
9.2.2	GPS 的介绍	(306)
9.2.3	GARMIN GPS 25LP 介绍	(307)
9.2.4	硬件设计	(308)
9.2.5	软件设计	(309)
9.3	USB 总线接口设计	(314)
9.3.1	实例说明	(314)
9.3.2	USB 简介	(315)
9.3.3	USB 接口芯片 PDIUSBD12 介绍	(316)
9.3.4	硬件设计	(319)
9.3.5	软件设计	(320)
9.4	基于 RTL8019AS 的以太网接口实验	(335)
9.4.1	实例说明	(335)
9.4.2	设计思路分析	(336)
9.4.3	以太网协议	(336)
9.4.4	芯片概述	(338)
9.4.5	硬件电路设计	(348)
9.4.6	软件设计	(350)
9.5	低频信号发生器输出	(354)
9.5.1	实例说明	(354)
9.5.2	DAC0832 介绍	(354)
9.5.3	硬件设计	(359)
9.5.4	软件设计	(360)
9.6	基于 8255A 芯片的微型打印机接口	(366)
9.6.1	实例说明	(366)
9.6.2	8255A 介绍	(366)
9.6.3	硬件设计	(370)
9.6.4	软件设计	(371)

9.7 单片机实现智能电热水器设计..... (371)

9.7.1 实例效果说明..... (372)

9.7.2 水温与流量、加热功率的关系..... (372)

9.7.3 硬件设计 (372)

9.7.4 软件设计 (375)

9.8 红外遥控器的设计 (385)

9.8.1 实例效果说明..... (386)

9.8.2 系统框图 (386)

9.8.3 硬件电路的设计..... (387)

9.8.4 软件设计 (390)

附录 A C51 库函数 (399)

附录 B 语法信息..... (403)

B.1 致命错误信息..... (403)

B.2 语法和语义错误信息 (404)

参考文献 (414)

第 1 章 C51 单片机基础

单片机的全称是单片微型计算机，是一种单硅片上集成微型计算机主要功能部件的集成芯片。C51 单片机在我国拥有庞大的用户群体。本章将重点介绍 C51 单片机的硬件基础知识。

1.1 C51 单片机基本介绍

在 1970 年微处理器研制成功之后，就出现了单片机。1976 年 9 月 Intel 公司的 MCS-48 单片机问世，在这短短的十几年内，单片机平均每二三年便要更新一代，其集成度也增加了一倍，功能更是翻了一番。

C51 系列 8 位单片机是在 20 世纪 80 年代初推出的新产品，它的主要的技术特征是扩大了片内存储容量、外部寻址空间，程序存储器和外部数据存储器的寻址都增加为 64 KB。内部程序存储器为 4 KB×8 ROM，用来存放系统程序、用户程序和固定常数。

8031、8751 与 8051 都属于 C51 单片机，它们的内部结构基本相同，区别在于 8031 内部不含有程序存储器，必须由外部扩展。8751 内部程序存储器为可编程、可擦写的只读存储器 EPROM，其内部程序由用户自行写入。

片内数据存储器采用 8 位地址，寻址范围为 256 B，其中 00H~7FH 为 128 B 的内部 RAM，用来存放用户的随机数；在 80H~FFH 范围内离散地分布着 21 个特殊功能寄存器，其中的 11 个特殊功能寄存器具有位寻址能力。在内部 RAM 中，00H~1FH 可分为 4 个寄存器工作区，由选择指令来切换寄存器工作区，从而有效地提高了 CPU 的现场保护能力和实时响应速度。20H~2FH 单元可进行位寻址。

并行口增强，并增设了全双工串行口 I/O：4 个 8 位并行 I/O 接口可用于地址和数据的传送，也可与 8243、8155 等连接，来扩展外部 I/O 接口。串行 I/O 接口是一个全双工串行通信接口，可用于数据的串行接收和发送。

两个定时/计数器均为 16 位（比 8048 多 1 倍），有 4 种工作方式，提高了定时/计数范围，并让用户使用更加灵活方便。

当 C51 的单片微型计算机进入到 80C51 的 MCU 时代后，就形成了可满足大量嵌入式应用的单片机系列产品。基于 Flash ROM 的 ISP/IAP 技术，改变了单片机应用系统的结构模式以及开发和运行条件。Atmel 公司的 AT89C×× 系列最早实现了 Flash ROM 技术。

将各厂家生产的与 51 兼容的单片机系列都称为 80C51 系列，它们都采用 CMOS 工艺，并与 C51 兼容。80C51 系列单片机保留了 C51 单片机的所有特性，内部组成基本相同，并增设了两种可以用软件进行选择的低功耗工作方式：空闲方式和掉电方式，其最主要技术特点是向外部接口电路扩展，以实现微控制器（microcontroller）完善的控制功能，另外还增加了一些外部接口功能单元，如 A/D、PWM、WDT（监视定时器）、高速 I/O 接口、PCA（可编程计数器阵列）、计数器的捕获/比较逻辑等。

1.1.1 引脚功能说明

80C51 的引脚功能如图 1-1 所示。

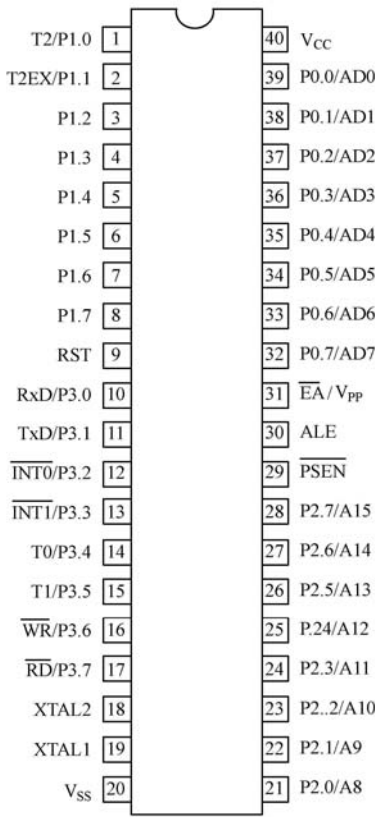


图 1-1 80C51 引脚功能图

80C51 的引脚功能说明如表 1-1 所示。

表 1-1 引脚功能说明

引 脚 名 称	引脚号 (DIP)	类 型	功 能 说 明
V _{ss}	20	I	接地
V _{cc}	40	I	供电电压
P0.0~P0.7	39~32	I/O	P0 口为一个 8 位漏极开路双向 I/O 口，每个引脚可吸收 8 个 TTL 门电流。当 P1 口的引脚第一次写“1”时，被定义为高阻输入。P0 能够用于外部程序数据存储器，可以被定义为数据/地址的第 8 位。在 Flash 编程时，P0 口作为原码输入/输出，当 Flash 进行校验时，P0 输出原码，此时 P0 外部必须被拉高
P1.0~P1.7	1~8	I/O	P1 口是一个内部提供上拉电阻的 8 位双向 I/O 口，P1 口缓冲器能接收输出 4 个 TTL 门电流。P1 口引脚写入“1”后，被内部上拉为高，可用做输入，P1 口被外部下拉为低电平时，将输出电流，这是由于内部上拉的原因。在 Flash 编程和校验时，P1 口作为第 8 位地址接收

引脚名称	引脚号 (DIP)	类型	功能说明
P2.0~P2.7	21~28	I/O	P2 口为一个内部上拉电阻的 8 位双向 I/O 口, P2 口缓冲器可接收、输出 4 个 TTL 门电流, 当 P2 口被写“1”时, 其引脚被内部上拉电阻拉高, 且作为输入。作为输入时, P2 口的引脚被外部拉低, 将输出电流, 这是由于内部上拉的原因。当 P2 口用于外部程序存储器或 16 位地址外部数据存储器进行存取时, P2 口输出地址的高 8 位。在给出地址“1”时, 利用内部上拉电阻, 当对外部 8 位地址数据存储器进行读/写时, P2 口输出其特殊功能寄存器的内容。P2 口在 Flash 编程和校验时接收高 8 位地址信号和控制信号
P3.0~P3.7	10~17	I/O	P3 口是 8 个带内部上拉电阻的双向 I/O 口, 可接收、输出 4 个 TTL 门电流。当 P3 口写入“1”后, 它们被内部上拉电阻拉高为高电平, 并用做输入, 作为输入时, 由于外部下拉为低电平, P3 口将输出电流 (ILL), 这是由于上拉电阻的原因
RST	9	I	复位输入。当振荡器复位器件时, 要保持 RST 引脚两个机器周期的高电平时间
ALE	30	O	当访问外部存储器时, 地址锁存允许的输出电平用于锁存地址的低位字节。在 Flash 编程期间, 此引脚用于输入编程脉冲。通常, ALE 端以不变的频率周期输出正脉冲信号, 此频率为振荡器频率的 1/6, 因此它可用做对外部输出的脉冲或用做定时目的。要注意的是: 当用做外部数据存储器时, 将跳过一个 ALE 脉冲。如果想禁止 ALE 的输出, 可在 SFR 8EH 地址上置“0”。此时, ALE 只有在执行 MOVX 和 MOVC 指令时 ALE 才起作用。另外, 该引脚被略微拉高。如果微处理器在外部执行状态 ALE 禁止, 置位无效
$\overline{\text{PSEN}}$	29	O	外部程序存储器的选通信号。在外部程序存储器取值期间, 每个机器周期两次 $\overline{\text{PSEN}}$ 有效。但在访问外部数据存储器时, 将不会出现两次有效的 $\overline{\text{PSEN}}$ 信号
$\overline{\text{EA}}/\text{V}_{\text{PP}}$	31	I	当 $\overline{\text{EA}}$ 保持低电平时, 则在此期间外部程序存储器的地址为 0000H~FFFFH, 无论是否有内部程序存储器。加密方式“1”时, $\overline{\text{EA}}$ 将内部锁定为 RESET; 当 $\overline{\text{EA}}$ 端保持高电平时, 此间内部程序存储器。在 Flash 编程期间, 此引脚也用于施加 12V 编程电源端 (V_{PP})
XTAL1	19	I	反向振荡放大器的输入和内部时钟工作电路的输入
XTAL2	18	O	反向振荡器的输出

P3 口也可以作为 89C51 的一些特殊功能口, 其功能如表 1-2 所示。

表 1-2 P3 口功能说明

引脚名称	特殊功能	引脚名称	特殊功能
P3.0 RXD	串行输入口	P3.4 T0	计时器 0 外部输入
P3.1 TXD	串行输出口	P3.5 T1	计时器 1 外部输入
P3.2 $\overline{\text{INT0}}$	外部中断 0	P3.6 $\overline{\text{WR}}$	外部数据存储器写选通
P3.3 $\overline{\text{INT1}}$	外部中断 1	P3.7 $\overline{\text{RD}}$	外部数据存储器读选通

1.1.2 C51 单片机的特点

8051 单片机包含数据总线、地址总线和控制总线三条总线及中央处理器、程序存储器 (ROM)、数据存储器 (RAM)、定时/计数器、并行接口、串行接口和中断系统等几个单元。C51 单片机的基本结构如图 1-2 所示。

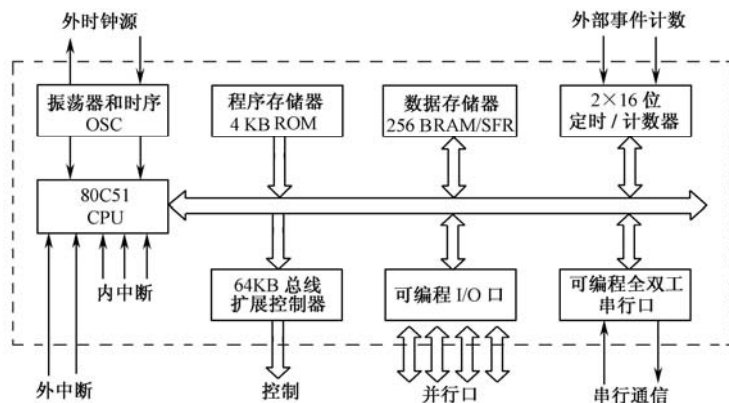


图 1-2 8051 单片机的基本结构示意图

C51 单片机是 Intel 公司推出的通用型单片机，基本的型号有 8051、8031 和 8751，它们除了 RAM 的类型不同以外，其他性能是完全相同。

C51 单片机基本特性如下：

- 8 位的 CPU；
- 片内有振荡器和时钟电路，工作频率为 1~12 MHz；
- 片内有 128/256 B RAM；
- 片内有 4 KB 程序存储器 ROM；
- 64 KB 片外数据存储器 RAM；
- 64 KB 片外程序存储器 ROM；
- 片内 21 个特殊功能寄存器 (SFR)；
- 4 个 8 位的并行 I/O 口 (PIO)；
- 1 个全双工串行口 (SIO/UART)；
- 2/3 个 16 位定时/计数器 (TIMER/COUNTER)；
- 可处理 5/6 个中断源，两级中断优先级；
- 内置布尔处理机 (位处理机)；
- C51 指令集包含 111 条指令。

C51 单片机的产品种类很多，配置如表 1-3 所示。

表 1-3 C51 单片机配置

系 列	片内存储器				定时/ 计数器	并行 I/O	串行 I/O	中 断 源
	片内 ROM			片内 RAM				
	无	有 ROM	有 EPROM					
Intel C51 子系列	8031	8051	8751	128 B	2×16	4×8 位	1	5
	80C31	80C51 (4 KB)	87C51 (4 KB)					
	8032	8052	8752	256 B	3×16	4×8 位	1	6
	80C32	80C52 (8 KB)	87C52 (8 KB)					
Intel C52 子系列	1051(1 KB)/ 2051(2 KB)/ 4051(4 KB) (20 引脚 DIP 封装)			128 B	2	15	1	5
	89C51(4 KB)/ 89C52(8 KB) (40 引脚 DIP 封装)			128/256 B	2/3	32	1	5/6

1.2 C51 单片机的内部结构

C51 单片机的内部结构可以划分为 CPU、存储器、并行口、串行口、定时/计数器、中断逻辑几部分。

1.2.1 CPU

C51 的 CPU 由运算器和控制逻辑组成。采用 SFR 集中控制，产生各种控制信号，用来控制存储器、I/O 口的数据传送、数据运算及位处理等操作。

C51 单片机的 CPU 内部结构如图 1-3 所示。

从图 1-3 中可看出，虚线框内的是 CPU 的内部结构，包含算术逻辑单元 ALU（arithmetic logic unit，也称为运算器）、累加器 A（8 位）、寄存器 B（8 位）、程序状态字 PSW（8 位）、程序计数器 PC（有时也称为指令指针，即 IP，16 位）、地址寄存器 AR（16 位）、数据寄存器 DR（8 位）、指令寄存器 IR（8 位）、指令译码器 ID、控制器等部件。

1. 运算器（ALU）

由于 ALU 内部没有寄存器，参加运算的操作数，必须放在累加器 A 中。累加器 A 也用于存放运算结果。

例如：执行指令 ADD A, B

执行这条指令时，累加器 A 中的内容通过输入口 In_1 输入 ALU，寄存器 B 通过内部数据总线经输入口 In_2 输入 ALU，A+B 的结果通过 ALU 的输出口 Out、内部数据总线，送回到累加器 A。

算术逻辑单元 ALU 的功能如下：

① 算术和逻辑运算，可对半字节（一个字节是 8 位，半个字节就是 4 位）和单字节数据进行操作；

- ② 加、减、乘、除、加 1、减 1、比较等算术运算；
- ③ 与、或、异或、求补、循环等逻辑运算；
- ④ 位处理功能（即布尔处理器）。

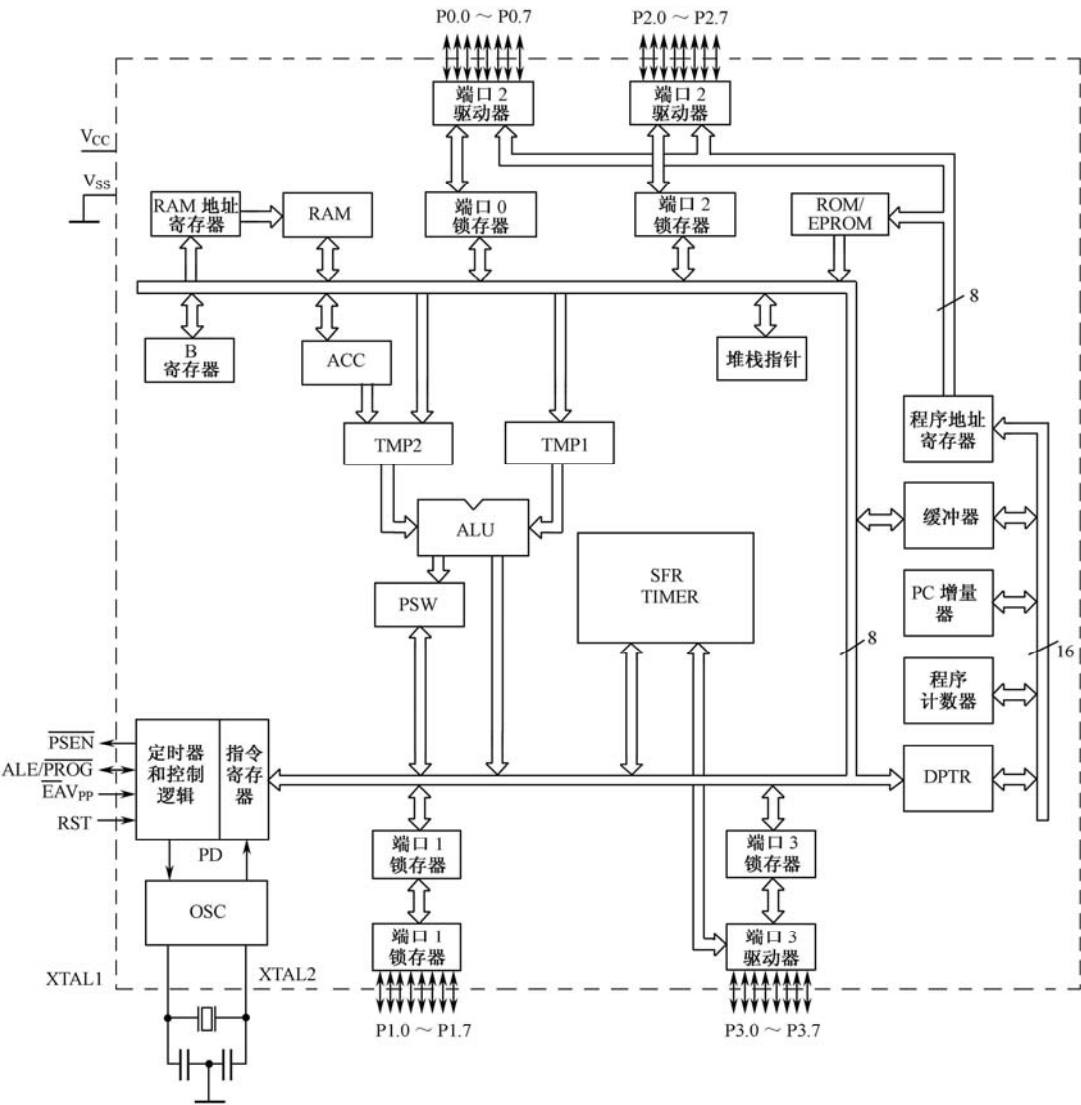


图 1-3 C51 单片机的 CPU 内部结构示意图

程序状态字寄存器 PSW 是 8 位寄存器，也称为标志寄存器，存放各有关标志，其格式和定义如表 1-4 所示。

表 1-4 程序状态字寄存器 PSW 格式

位编号	PSW.7	PSW.6	PSW.5	PSW.4	PSW.3	PSW.2	PSW.1	PSW.0
位地址	D7H	D6H	D5H	D4H	D3H	D2H	D1H	D0H
位定义名	Cy	AC	F0	RS1	RS0	0V	F1	P

程序状态字寄存器 PSW 各位的定义如下:

- Cy: 进位标志。用于表示 Acc.7 是否有向更高位进位或借位, 有进位或者借位时, Cy=1, 否则 Cy=0;
- AC: 辅助进位标志, 用于表示 Acc.3 是否有向 Acc.4 进位, 常用于十进制调整运算中;
- RS1、RS0: 工作寄存器区选择控制位;
RS1、RS0 = 00 —— 0 区 (00H~07H);
RS1、RS0 = 01 —— 1 区 (08H~0FH);
RS1、RS0 = 10 —— 2 区 (10H~17H);
RS1、RS0 = 11 —— 3 区 (18H~1FH);
- OV: 溢出标志, 表示 Acc 在有符号数算术运算中的溢出;
- P: 奇偶标志, 表示 Acc 中“1”的个数的奇偶性;
- F0、F1: 用户标志。

地址寄存器 AR (16 位) 的作用是用来存放将要寻址的外部存储器单元的地址信息, 指令码所在存储单元的地址编码, 由程序计数器 PC 产生, 而指令中操作数所在的存储单元地址码是由指令的操作数给定的。

数据寄存器 DR 用于存放写入外部存储器或 I/O 端口的数据信息。可见, 数据寄存器对输出数据具有锁存功能。数据寄存器与外部数据总线 DB 是直接相连的。

2. 控制器

控制器一般包括时钟电路、复位电路、定时控制逻辑、指令寄存器 IR、指令译码器 ID、程序计数器 PC 和信息传送控制部件等。C51 的控制器在单片机内部协调各功能部件之间的数据传送、数据运算等操作, 并对单片机发出控制信息。它以主振频率为基准, 由定时控制逻辑发出 CPU 时序, 将指令寄存器中存放的指令码取出送到指令译码器进行译码, 再由信息传送控制部件发出一系列的控制信号, 控制单片机各部分的运行, 完成指令指定的功能。

(1) 时钟电路

时钟电路用于产生单片机工作所需的时钟信号。单片机在时钟信号控制下, 各部件之间协调一致地工作, 时钟信号控制着计算机的工作节奏。8051 内置的时钟电路最高频率达 12 MHz, 用于产生整个单片机运行的脉冲时序, 但 8051 单片机需要外接振荡电容。

C51 单片机内部由一个反向放大器构成振荡器, 可以由它产生时钟, 图 1-4 是 8051 内部时钟电路图。

在图 1-4 中, 引脚 XTAL1 和 XTAL2 之间跨接的晶体振荡器 (晶振) 和微调电容, 可以和芯片内部的反相放大器构成一个稳定的自激振荡器, 这就产生了时钟, 这种方式称为内部时钟源方式。电容器 C_1 和 C_2 的主要作用是帮助振荡器起振, 且电容器大小对振荡频率有微调作用, 典型值为 $C_1=C_2=30\text{ pF}$ 。振荡频率主要由石英晶振的频率确定, 当石英晶振的频率为 6 MHz 时, 时钟频率为 3 MHz。目前, C51 系列单片机的晶振频率范围 f_{osc} 为 1.2~60 MHz, 其典型值为 6 MHz、12 MHz、11.0592 MHz、20 MHz、24 MHz、33 MHz、40 MHz 等。

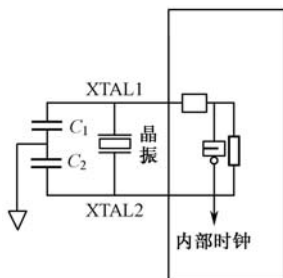


图 1-4 8051 内部时钟电路图

8051 的工作时钟也可以由外部时钟信号提供，外部时钟电路如图 1-5 所示。

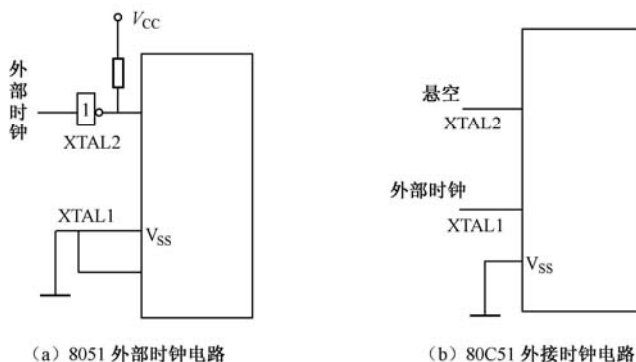


图 1-5 8051 外部时钟电路图

在图 1-5 (a) 中，外部的时钟信号由引脚 XTAL2 引入。由于引脚 XTAL2 的逻辑电平不是 TTL 的，故需外接一上拉电阻，外接的时钟频率应低于 12 MHz。

图 1-5 (b) 是 80C51 的外部时钟接法，其外部时钟信号由引脚 XTAL1 引入，而引脚 XTAL2 悬空。

(2) CPU 时序

所谓时序是指各种信号的时间序列，它表明了指令执行中各种信号之间的相互关系。单片机本身就是一个复杂的时序电路，CPU 执行指令的一系列动作都是在时序电路控制下一拍一拍进行的。为了达到同步协调工作的目的，各操作信号在时间上有严格的先后次序，这些次序就是 CPU 的时序。

CPU 的时序信号有两大类：一类用于单片机内部，控制片内各功能部件；另一类时序信号通过控制总线送到片外，这类时序信号在系统扩展中比较重要，应当了解。

8051 单片机以晶体振荡器的振荡周期（或外部引入的时钟信号的周期）为最小的时序单位，所以片内的各种微操作都是以晶振周期为时序基准。8051 单片机的时钟信号图如图 1-6 所示。

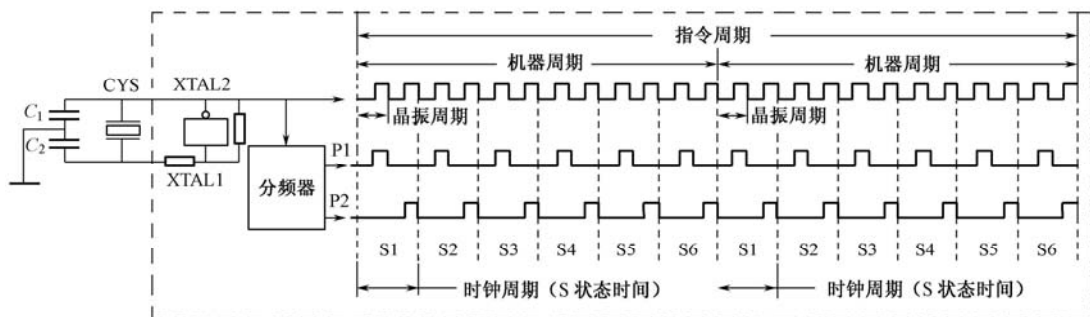


图 1-6 8051 单片机的时钟信号

由图中可以看出，8051 单片机的基本定时单位共有 4 个，它们从小到大分别是：

- ① 晶振周期是由振荡电路产生的振荡脉冲的周期，又称节拍（如 P1、P2）。
- ② 时钟周期是晶振周期的 2 倍，即一个时钟周期包含两个相互错开的节拍，也称 S 状态时间。

③ 机器周期。C51 单片机有固定的机器周期，它是由晶振频率经过 12 分频后形成的，也就是说，一个机器周期是晶振周期的 12 倍。

单片机的基本操作周期为机器周期。一个机器周期有 6 个状态，每个状态由两个脉冲（晶振周期）组成，即 1 个机器周期=6 个状态周期=12 个晶振周期。

若单片机采用 12 MHz 的晶体振荡器，则一个机器周期为 1 μs，若采用 6 MHz 的晶体振荡器，则一个机器周期为 2 μs。

④ 指令周期是指执行一条指令所需要的时间。不同的指令，其执行时间各不相同，如果用占用机器周期多少来衡量的话，C51 单片机的指令可分为单周期指令、双周期指令及四周期指令。

单片机的指令的执行过程可以分为取指令、译码、执行三个过程。取指令的过程实质上是访问程序存储器的过程，其时间的长短取决于指令的字节数；译码与执行时间的长短取决于指令的类型。对于 C51 单片机的指令系统，其指令长度为 1~3 B，其中单字节指令的运行时间有单机器周期、双机器周期和四机器周期；双字节指令有双字节单机器周期指令和双字节双机器周期指令；三字节指令则都为双机器周期指令。通常算术和逻辑操作在节拍 P1 期间进行，内部寄存器的传送在节拍 P2 期间进行。下面就简单介绍几个典型的时序。

对于单机器周期指令，是在 S1P2 时刻把指令读入指令寄存器，并开始执行指令，在 S6P2 结束时完成指令操作。中间在 S4P2 时刻读的下一条指令要丢弃，且程序计数器 PC 也不加 1，如图 1-7 所示。

对于双字节单机器周期指令，则在同一机器周期的 S4 和 P2 时刻将第二个字节读入指令寄存器，并开始执行指令，在 S6 和 P2 时刻结束该指令的操作，如图 1-8 所示。

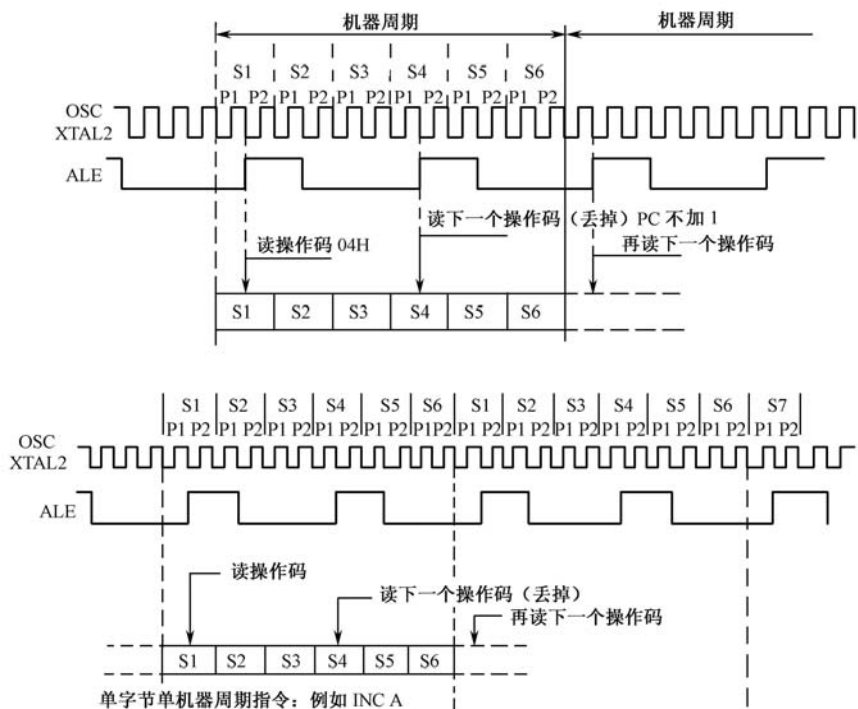


图 1-7 单机器周期指令

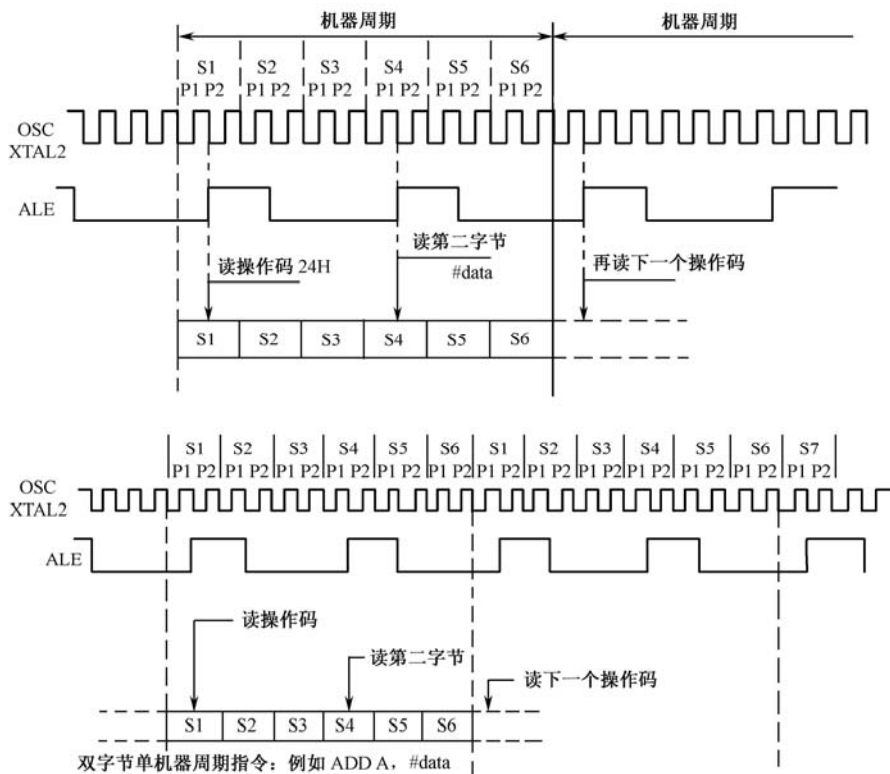


图 1-8 双字节单机器周期指令

对于单字节双机器周期指令，在 2 个机器周期内要发生 4 次读操作码的操作，由于是单字节指令，后 3 次读操作都无效，如图 1-9 所示。

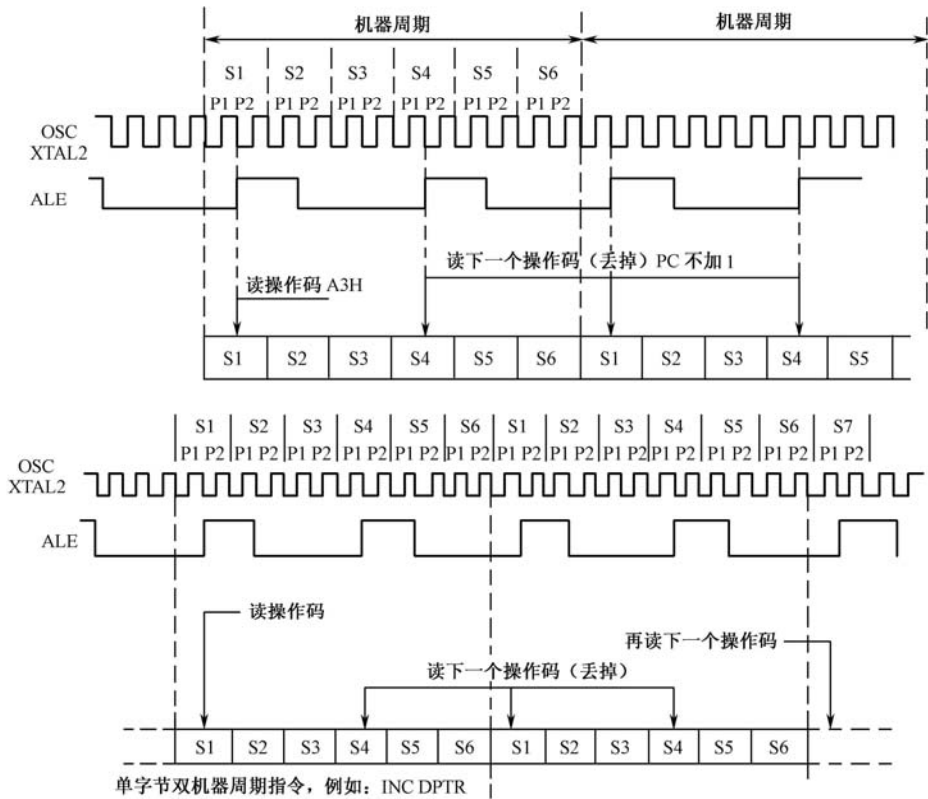


图 1-9 单字节双机器周期指令

但访问外部数据存储器指令 `MOVX` 的时序有所不同，它也是单字节双机器周期指令，在第 1 个机器周期有 2 次读操作，后一次无效，从 S5 时刻开始送出外部数据存储器的地址，随后进行读/写数据，在读/写期间 ALE 端不产生有效信号。在第 2 个机器周期，不发生读操作，如图 1-10 所示。

该部分操作涉及到的指令部件如下。

程序计数器 PC 不属于特殊功能寄存器，不可访问，在物理结构上是独立的。PC 的作用是用来存放将从 ROM 中读出的下一字节指令码的地址，共 16 位，可对 64 KB ROM 进行直接寻址，PC 低 8 位经 P0 口输出，高 8 位经 P2 口输出。

PC 的基本工作方式如下：

- 自动加 1，CPU 从 ROM 中每读一个字节，自动执行 $PC+1 \rightarrow PC$ ；
- 执行转移指令时，PC 会根据该指令要求修改下一次读 ROM 新的地址；
- 执行调用子程序或发生中断时，CPU 会自动将当前 PC 值压入堆栈，将子程序入口地址或中断入口地址装入 PC；子程序返回或中断返回时，恢复原来被压入堆栈的 PC 值，继续执行原顺序程序指令。

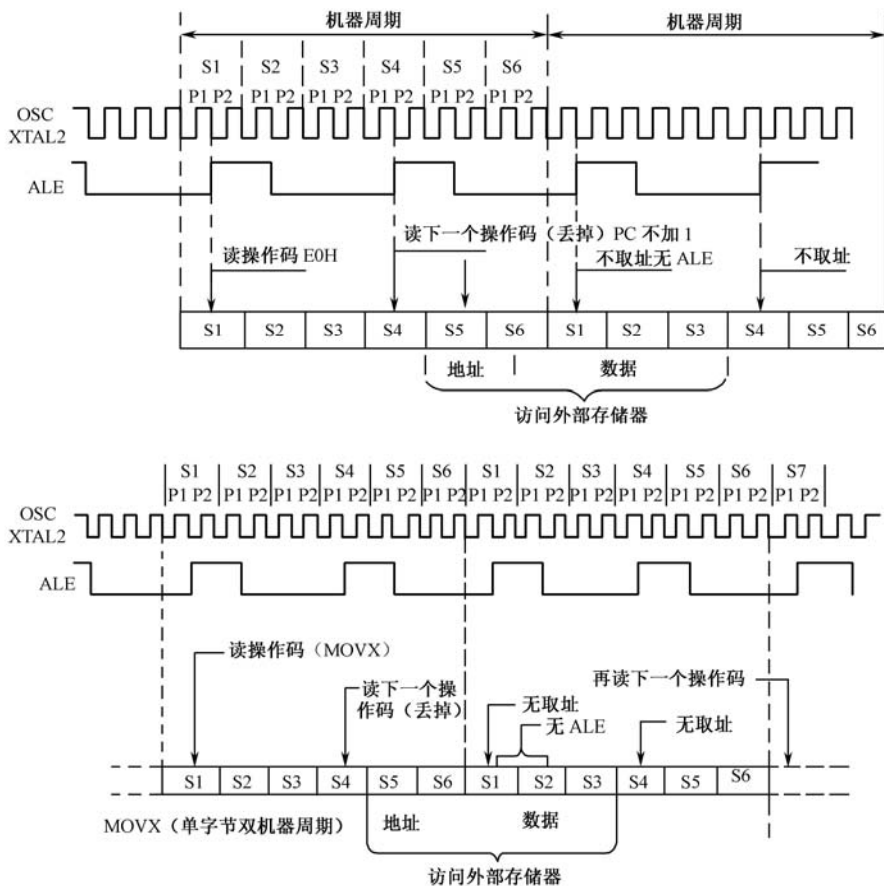


图 1-10 MOVX (单字节双机器周期)

指令寄存器 IR 的作用是用来存放即将执行的指令代码。在这里先简单地了解 CPU 执行指令的过程，首先由程序存储器 (ROM) 中读取指令代码送入到指令寄存器，经译码器译码后再由定时与控制电路发出相应的控制信号，从而完成指令的功能。

指令译码器 ID 用于对送入指令寄存器中的指令进行译码，所谓译码就是把指令转变成执行此指令所需要的电信号。当指令送入译码器后，由译码器对该指令进行译码，根据译码器输出的信号，CPU 控制电路定时地产生执行该指令所需的各种控制信号，使单片机正确地执行程序所需要的各种操作。

数据指针 DPTR 是 16 位地址寄存器，既可以用于寻址外部数据寄存器，也可以寻址外部程序存储器中的表格数据。DPTR 可以寻址 64 KB 的地址空间。

1.2.2 存储器结构

80C51 单片机的存储器配置方式与其他常用的微机系统不同，属于哈佛结构型的单片机，它把程序存储器和数据存储器分开，有各自的寻址系统、控制信号和功能。程序存储器用于存放程序和表格常数，数据存储器用于存放程序运行数据和结果。

8051 单片机的存储器在物理结构上分为程序存储器空间和数据存储器空间，共有 4 个

存储空间：片内程序存储器、片外程序存储器以及片内数据存储器、片外数据存储器。但从用户的角度，8051 存储器地址空间可分为 3 类：片内、片外统一编址 0000H~FFFFH 的 64 KB 程序存储器地址空间（用 16 位地址）；64 KB 片外数据存储器地址空间，地址也从 0000H~FFFFH（用 16 位地址）编址；256 B 片内数据存储器地址空间（用 8 位地址）。上述 4 个存储空间地址是重叠的，如图 1-11 所示。

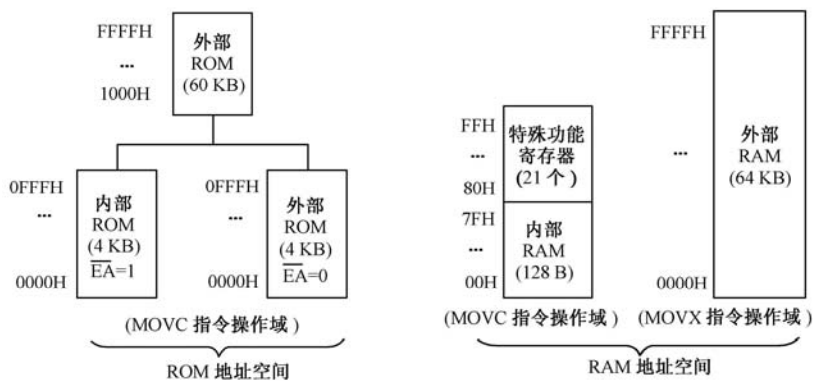


图 1-11 C51 单片机存储空间配置图

8051 的指令系统设计了不同的数据传送指令以区别这 4 个不同的存储空间：CPU 访问片内、片外 ROM 指令用 MOVC，访问片外 RAM 指令用 MOVX，访问片内 RAM 指令用 MOV。

1. 程序存储器

程序存储器用于存放已经编写好的程序和表格常数。程序相当于给微处理器处理问题的一系列命令，它和数据在本质上是一样的，都是由机器码组成的代码串，只是程序代码存放于程序存储器中。程序通过 16 位程序计数器寻址，寻址能力为 64 KB，这使得指令能在 64 KB 的地址空间内任意跳转，但不能使程序从程序存储器空间转移到数据存储器空间。

不同类型的单片机程序存储器的配置也不同，如表 1-5 所示。

表 1-5 单片机程序存储器的配置

单片机类型	存储器类型	地址范围
8051	内部有 4 KB ROM/EPROM	0000H~0FFFH
	外部的 ROM/EPROM	1000H~FFFFH
52 子系列	内部 8 KB ROM/EPROM	0000H~1FFFH
	外部的 ROM/EPROM	2000H~FFFFH
8751	内部有 4 KB ROM/EPROM	0000H~0FFFH
	外部的 ROM/EPROM	1000H~FFFFH
8031/8032	内部没有 ROM/EPROM	0000H~FFFFH

C51 单片机具有 64 KB 程序存储器寻址空间，用于存放用户程序、数据和表格等信息。对于内部无 ROM 的 8031 单片机，它的程序存储器必须由外部扩展，空间地址为 64 KB，

此时单片机的EA端必须接地，强制CPU从外部程序存储器读取程序。对于内部有ROM的8051等单片机，当引脚EA接高电平时，程序计数器PC执行片内ROM中的程序，当指令地址超过片内ROM地址时，就自动转向片外ROM中去取指令；当引脚EA接低电平（接地）时，8051片内ROM不起作用，CPU只能从片外ROM中取指令，地址可以从0000H开始编址。8051从片内程序存储器和片外程序存储器取指时的执行速度相同。

8051单片机片内有4KB的程序存储单元，其地址为0000H~0FFFH，单片机启动复位后，程序计数器的内容为0000H，所以系统将从0000H单元开始执行程序。

但在程序存储中有些特殊的单元，这在使用中应加以注意。

其中一组特殊单元是0000H~0002H，系统复位后，PC为0000H，单片机从0000H单元开始执行程序，如果程序不是从0000H单元开始，则应在这三个单元中存放一条无条件转移的指令，让CPU直接去执行用户指定的程序。

另一组特殊单元是0003H~002AH，这40个单元各有用途，它们被均匀地分为五段，每段的定义如下：

- 0003H~000AH 外部中断0中断地址区；
- 000BH~0012H 定时/计数器0中断地址区；
- 0013H~001AH 外部中断1中断地址区；
- 001BH~0022H 定时/计数器1中断地址区；
- 0023H~002AH 串行中断地址区。

以上的40个单元是专门用于存放中断处理程序的地址单元。在中断响应后，按中断的类型，自动转到各自的中断区去执行程序，因此以上地址单元不能用于存放程序的其他内容，只能存放中断服务程序。但是，在通常情况下，每段只有8个地址单元是不能存下完整的中断服务程序的，因而一般也在中断响应的地址区安放一条无条件转移的指令，指向程序存储器的其他真正存放中断服务程序的空间去执行，这样在中断响应后，CPU就能读到这条转移指令，便转向其他地方去继续执行中断服务程序。

2. 数据存储器

数据存储器也称为随机存取数据存储器。C51单片机的数据存储器在物理上和逻辑上都分为两个地址空间，一个是内部数据存储器区和一个外部数据存储器区。C51单片机内部RAM

有128B或256B的用户数据存储（不同的型号大小不同），它们是用于存放执行的中间结果和过程数据的。C51单片机的数据存储器均可读/写，部分单元还可以位寻址。

8051单片机内部RAM共有256个单元，这256个单元共分为两部分。其一是地址从00H~7FH单元（共128B）为用户数据RAM；从80H~FFH地址单元（也是128B）为特殊寄存器（SFR）单元，如图1-12所示，在图中可清楚地看出它们的结构分布。

00H~1FH共32个单元被均匀地分为4块，每块包

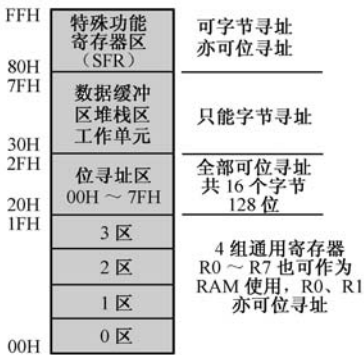


图 1-12 数据存储器结构分布

含 8 个 8 位寄存器，均以 R0~R7 来命名，通常称这些寄存器为通用寄存器。这 4 块中的寄存器都称为 R0~R7，那么在程序中怎么区分和使用它们呢？Intel 工程师们又安排了一个寄存器——程序状态字寄存器（PSW）来管理它们，CPU 只要定义这个寄存的 PSW 的第 3 和第 4 位（RS0 和 RS1），即可选中这四组通用寄存器。

内部 RAM 的 20H~2FH 单元为位寻址区，既可作为一般单元用字节寻址，也可对它们的位进行寻址。位寻址区共有 16 B，128 位，位地址为 00H~7FH。位地址分配如表 1-6 所示。

表 1-6 RAM 位寻址区地址表

单 元 地 址	MSB位地址LSB							
2FH	7FH	7EH	7DH	7CH	7BH	7AH	79H	78H
2EH	77H	76H	75H	74H	73H	72H	71H	70H
2DH	6FH	6EH	6DH	6CH	6BH	6AH	69H	68H
2CH	67H	66H	65H	64H	63H	62H	61H	60H
2BH	5FH	5EH	5DH	5CH	5BH	5AH	59H	58H
2AH	57H	56H	55H	54H	53H	52H	51H	50H
29H	4FH	4EH	4DH	4CH	4BH	4AH	49H	48H
28H	47H	46H	45H	44H	43H	42H	41H	40H
27H	3FH	3EH	3DH	3CH	3BH	3AH	39H	38H
26H	37H	36H	35H	34H	33H	32H	31H	30H
25H	2FH	2EH	2DH	2CH	2BH	2AH	29H	28H
24H	27H	26H	25H	24H	23H	22H	21H	20H
23H	1FH	1EH	1DH	1CH	1BH	1AH	19H	18H
22H	17H	16H	15H	14H	13H	12H	11H	10H
21H	0FH	0EH	0DH	0CH	0BH	0AH	09H	08H
20H	07H	06H	05H	04H	03H	02H	01H	00H

CPU 能直接寻址这些位，执行例如置 1、清 0、求反、转移，传送和逻辑等操作。常称 C51 单片机具有布尔处理功能，布尔处理的存储空间指的就是这些为寻址区。

3. 特殊功能寄存器

特殊功能寄存器（SFR）也称为专用寄存器，它反映了 C51 单片机的运行状态，很多功能也通过特殊功能寄存器来定义和控制程序的执行。

C51 单片机有 21 个特殊功能寄存器，它们被离散地分布在内部 RAM 的 80H~FFH 地址中，这些寄存的功能已做了专门的规定，用户不能修改其结构。表 1-7 是特殊功能寄存器分布情况。

表 1-7 特殊功能寄存器

标 识 符 号	地 址	寄存器名称
ACC	0E0H	累加器
B	0F0H	B 寄存器

标识符号	地 址	寄存器名称
PSW	0D0H	程序状态字
SP	81H	堆栈指针
DPTR	82H、83H	数据指针（16 位）含 DPL 和 DPH
IE	0A8H	中断允许控制寄存器
IP	0B8H	中断优先控制寄存器
P0	80H	I/O 口 0 寄存器
P1	90H	I/O 口 1 寄存器
P2	0A0H	I/O 口 2 寄存器
P3	0B0H	I/O 口 3 寄存器
PCON	87H	电源控制及波特率选择寄存器
SCON	98H	串行口控制寄存器
SBUF	99H	串行数据缓冲寄存器
TCON	88H	定时控制寄存器
TMOD	89H	定时器方式选择寄存器
TL0	8AH	定时器 0 低 8 位
TH0	8CH	定时器 0 高 8 位
TL1	8BH	定时器 1 低 8 位
TH1	8DH	定时器 1 高 8 位

这里对其主要的寄存器进行简单的介绍。

（1）累加器 A（accumulator）

累加器 A 是一个最常用的专用寄存器，大部分单操作指令的一个操作数都取自累加器，很多双操作数指令中的一个操作数也取自累加器。加、减、乘、除法运算的指令和运算结果都存放于累加器 A 或 AB 累加器对中。大部分的数据操作都会通过累加器 A 进行，它如像一个交通要道，在程序比较复杂的运算中，累加器成了制约软件效率的“瓶颈”，它的功能较多，地位也十分重要。以致于后来发展的单片机，有的集成了多累加器结构，或者使用寄存器阵列来代替累加器，即赋予更多寄存器以累加器的功能，是为了解决累加器的“交通堵塞”问题，提高单片机的软件效率。

（2）寄存器 B

在乘除法运算指令中，乘法运算指令中的两个操作数分别取自累加器 A 和寄存器 B，其结果存放于 AB 寄存器对中。除法运算指令中，被除数取自累加器 A，除数取自寄存器 B，结果商存放于累加器 A，余数存放于寄存器 B 中。

（3）堆栈指针 SP（stack pointer）

堆栈是一种数据结构，它是一个 8 位寄存器，用来指示堆栈顶部在内部 RAM 中的位置。系统复位后，SP 的初始值为 07H，使得堆栈实际上是从 08H 开始的。但从 RAM 的结构分布中可知，08H~1FH 隶属 1~3 工作寄存器区，若编程时需要用到这些数据单元，必须对堆栈指针 SP 进行初始化，原则上设在任何一个区域均可，但一般设在 30H~1FH 之间较为适宜。

数据的写入堆栈的操作称为入栈（**PUSH**，有些文献也称为插入运算或压入），从堆栈中取出数据的操作称为出栈（**POP**，也称为删除运算或弹出），堆栈的最主要特征是“后进先出”，即最先入栈的数据放在堆栈的底部，而最后入栈的数据放在栈的顶部，因此，最后入栈的数据出栈的顺序是最先的，这和我们往一个箱里存放书本一样，要将最先放入箱底部的书取出，必须先取走最上层的书籍。堆栈结构如图 1-13 所示。

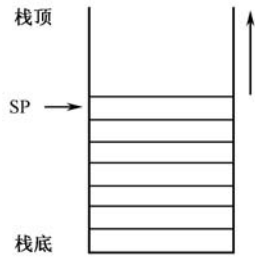


图 1-13 堆栈结构图

堆栈的设立是为了在中断操作和子程序的调用时用于保存数据的，即常说的断点保护和现场保护。微处理器无论是在转入子程序和中断服务程序的执行，执行完后，还是要返回到主程序中来的，在转入子程序和中断服务程序前，必须先将现场的数据保存起来，否则返回时，CPU 并不知道原来的程序执行到哪一步，所以在转入执行其他子程序前，先将需要保存的数据压入堆栈中保存，以备返回时，再复原当时的数据，供主程序继续执行。

转入中断服务程序或子程序时，需要保存的数据可能有若干个，都需要一一保留。如果微处理器进行多重子程序或中断服务程序嵌套，那么需保存的数据就比较多，这要求堆栈还需要有足够大的容量，否则会造成堆栈溢出，丢失应当备份的数据，轻则使运算和执行结果出错，重则使整个程序紊乱。

C51 单片机的堆栈是在 RAM 中开辟的，即堆栈要占据一定的 RAM 存储单元。同时 C51 单片机的堆栈可以由用户设置，SP 的初始值不同，堆栈的位置则不一定相同，不同的设计人员，可以使用不同的堆栈区，不同的应用要求，堆栈要求的容量也有所不同。堆栈的操作只有两种，即进栈和出栈，但不管是向堆栈写入数据还是从堆栈中读出数据，都是对栈顶单元进行的，SP 就是及时指示出栈顶的位置（即地址）。在子程序调用和中断服务程序响应的开始和结束期间，CPU 都是根据 SP 指示的地址与相应的 RAM 存储单元来交换数据的。

堆栈的操作有两种方法：一种是自动方式，即在中断服务程序响应或子程序调用时，返回地址自动进栈，当需要返回执行主程序时，返回的地址自动交给 PC，以保证程序从断点处继续执行，这种方式是不需要编程人员干预的；第二种方式是人工指令方式，使用专有的堆栈操作指令进行进出栈操作，也只有两条指令，进栈为 **PUSH** 指令，在中断服务程序或子程序调用时作为现场保护，出栈操作 **POP** 指令，用于子程序完成时，为主程序恢复现场。

（4）I/O 口专用寄存器（P0、P1、P2、P3）

I/O 口寄存器 P0、P1、P2 和 P3 分别是 C51 单片机的 4 组 I/O 口锁存器。C51 单片机并没有专门的 I/O 口操作指令，而是把 I/O 口也当做一般的寄存器来使用，数据传送都统一使用 **MOV** 指令来进行，这样的好处在于四组 I/O 口还可以当做寄存器直接寻址方式参与其他操作。

（5）定时/计数器（TL0、TH0、TL1 和 TH1）

C51 单片机中有两个 16 位的定时/计数器 T0 和 T1，它们由 4 个 8 位寄存器组成，而两个 16 位定时/计数器是完全独立的。可以单独对这四个寄存器进行寻址，但不能把 T0 和 T1 当做 16 位寄存来使用。

(6) 定时/计数器方式选择寄存器 (TMOD)

TMOD 寄存器是一个专用寄存器，用于控制两个定时计数器的工作方式，TMOD 可以用字节传送指令设置其内容，但不能位寻址，各位的定义如表 1-8 所示。

表 1-8 TMOD 寄存器

位序	D7	D6	D5	D4	D3	D2	D1	D0
位标志	GATE	C/ \overline{T}	M1	M0	GATE	C/ \overline{T}	M1	M0
定时/计数器 1					定时/计数器 0			

(7) 串行数据缓冲器 (SBUF)

串行数据缓冲器 SBUF 用来存放需要发送和接收的数据，它由两个独立的寄存器组成，一个是发送缓冲器，另一个是接收缓冲器，要发送和接收的操作其实都是对串行数据缓冲器进行。

1.2.3 片内并行接口

C51 单片机有 4 个 8 位的并行接口，分别为 P0、P1、P2 和 P3，共 32 根 I/O 线，每个接口主要由端口锁存器、输入缓冲器、输出驱动器和引至芯片外的端口引脚四部分组成。

1. 输入结构

I/O 口作为输入口时有两种工作方式，即所谓的读端口与读引脚。读端口时实际上并不从外部读入数据，而是把端口锁存器的内容读入到内部总线，经过某种运算或变换后再写回到端口锁存器，比如取反、置位、清 0 等指令；而读端口时才真正地把外部的数据读入到内部总线，图 1-14 中的两个三角形表示输入缓冲器，CPU 将根据不同的指令，分别发出读端口或读引脚信号，以完成不同的操作。C51 单片机端口的逻辑如图 1-14 所示。

读引脚时，也就是把端口作为外部输入线时，首先要通过外部指令把端口锁存器置 1，然后再实行读引脚操作，否则就可能读入出错。如果不对端口置 1，端口锁存器原来的状态有可能为 0 (Q 端为 0， \overline{Q} 为 1)，加到场效应管栅极的信号为 1，该场效应管就导通，对地呈现低阻抗，此时即使引脚上输入的信号为 1，也会因端口的低阻抗而使信号变为 0，使得外加的 1 信号读入后不一定是 1。若先执行置 1 操作，则可以使场效应管截止，引脚信号直接加到三态缓冲器中，实现正确的读操作。由于在输入操作时还必须附加一个准备动作，所以这类 I/O 口被称为准双向口。

2. 端口的工作原理

C51 单片机的 P0、P1、P2、P3 口作为输入/输出口时都是准双向口，除 P1 口外，P0、P2、P3 口都有第二功能。

(1) P0 口

P0 口的内部有一个 2 选 1 的选择器，受内部信号的控制，如果在图中的位置则处在 I/O 口工作方式，此时相当于一个准双向口。输入时须先将口置 1，每根接口线可以独立定义为输入/输出，但是须在接口线上加上拉电阻。如果将开关往另一个方向，则就是另一个功能，即作为地址/数据复用总线，此时不能逐位定义为输入/输出，它有两种用法：当做数据

总线用时，输入 8 位数据；而当做地址总线用时，则输出 8 位地址。再强调一点，当 P0 口作为地址/数据总线用之后，就再也不能作为 I/O 口使用了，即当单片机的并行口不够用时，需要扩展输入/输出口时的一种用法。

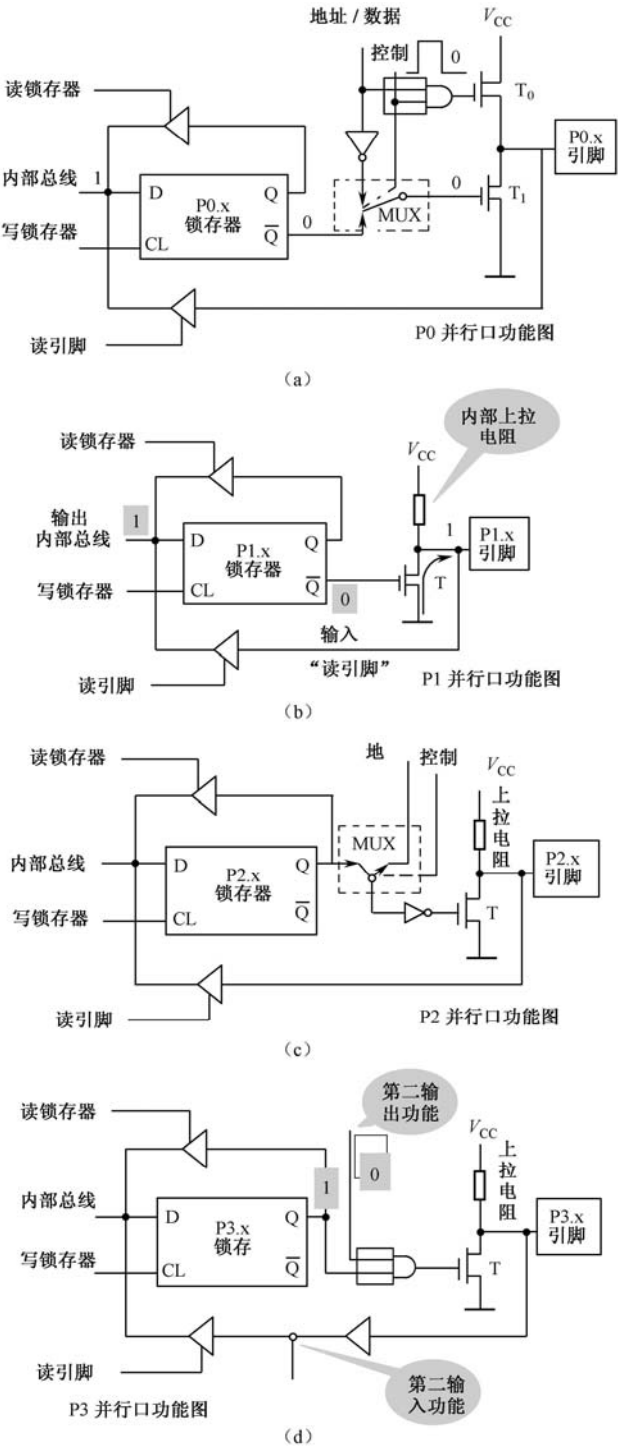


图 1-14 C51 单片机端口的逻辑图

(2) P1 口

与 P0 口不同，P1 口只能作为 I/O 口使用，但它的内部有一个上拉电阻，所以连接外围负载时不需要外接上拉电阻，P1、P2、P3 口在这一点都是一样的。

(3) P2 口

P2 口作为 I/O 口时，与 P0 口一样。当内部开关扳向另一个方向时，即作为地址输出时，可以输出程序存储器或外部数据存储器的高 8 位地址，并与 P0 口输出的低地址一起构成 16 位的地址线，从而可以分别寻址 64 KB 的程序存储器或外部数据存储器，同样地址线是 8 位一起自动输出的，不能像 I/O 口线那样逐位定义。

(4) P3 口

P3 口作为 I/O 口时，同 P1 口相同，也是准双向口。不同的是，P3 口的每一位都有另一种功能，也叫做第二功能，它们的具体作用如表 1-9 所示。

表 1-9 P3 口第二功能

端 口 引 脚	第 二 功 能	注 释
P3.0	RXD	串行口输入
P3.1	TXD	串行口输出
P3.2	INT0	外部中断 0
P3.3	INT1	外部中断 1
P3.4	T0	计数器 0 计数输入
P3.5	T1	计数器 1 计数输入
P3.6	WR	外部 RAM 写入选通信号
P3.7	RD	外部 RAM 读出选通信号

1.3 C51 单片机定时/计数器

定时/计数器和中断源的多少直接决定了单片机的性能。C51 单片机内部共有两个 16 位可编程的定时/计数器，即定时/计数器 T0 和定时/计数器 T1，它们既有定时功能又有计数功能。

1.3.1 定时/计数器结构

定时/计数器的基本部件是两个 8 位计数器（其中 TH1 和 TL1 是 T1 的定时/计数器，TH0 和 TL0 是 T0 的定时/计数器）。定时/计数器的基本结构如图 1-15 所示。

图中，TL0、TH0、TL1、TH1 以及 TMOD 和 TCON 都是单片机的特殊功能寄存器。TL0 和 TH0 组成 16 位的定时/计数器（T0），TL1 和 TH1 组成 16 位的定时/计数器（T1），TMOD 是 T0 和 T1 的工作方式控制寄存器，TCON 是 T0 和 T1 的运行状态控制寄存器。在实际应用中，应首先根据需要对这些寄存器进行初始化设置，即设置 T0 和 T1 的工作方式，并对 T0 和 T1 定时/计数器装入初始值以得到精确的定时时间。T0 和 T1 的工作方式及运行状态是由 TMOD 和 TCON 两个特殊功能寄存器控制的，而 TMOD 和 TCON 是由用户所编的程序控制的。

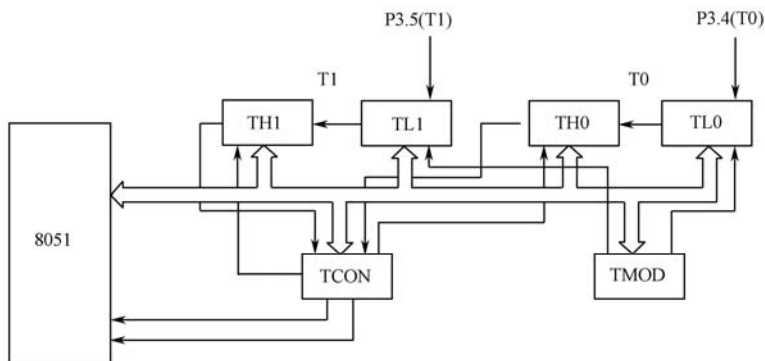


图 1-15 定时/计数器的基本结构

1.3.2 定时/计数器的方式控制字

单片机中的定时/计数器可以有两种用途，可以通过定时/计数器的方式控制字（实际上就是与定时/计数器有关的特殊功能寄存器）来设置，使它工作于需要的方式下，在单片机中有两个特殊功能寄存器与定时/计数器有关，它们分别是 **TMOD** 和 **TCON**。

TMOD 和 **TCON** 是特殊功能寄存器的名称，在写程序时既可以直接用这个名称来指定它们，也可以直接用地址 **89H** 和 **88H** 来指定它们。

1. 定时器控制寄存器（TCON）

定时器控制寄存器（**TCON**）各位定义如表 1-10 所示。

表 1-10 定时器控制寄存器（TCON）

作 用	用于定时/计数器				用于中断			
位地址	8FH	8EH	8DH	8CH	8BH	8AH	89H	88H
位符号	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TCON 被分成两部分，高 4 位用于定时/计数器，低 4 位则用于中断。当计数溢出后 **TF0** 就由 0 变为 1。**TR0** 为运行控制位，当要使用 **T0** 时，必须用指令 **SETB** 来置位以启动计数/定时器工作（用指令 **CLR** 可关闭定时/计数器的工作），仅当 **TR0** 为 1 时，开关才能合上，计数脉冲才能进入计数器。

2. 工作方式寄存器（TMOD）

工作方式寄存器（**TMOD**）用来确定定时器的工作方式及功能选择，不能位寻址。**TMOD** 各位的定义如表 1-11 所示。

表 1-11 工作方式寄存器（TMOD）

作 用	用于 T1				用于 T0			
位	D7	D6	D5	D4	D3	D2	D1	D0
位符号	GATE	C/\bar{T}	M_1	M_0	GATE	C/\bar{T}	M_1	M_0

TMOD 是 8 位的控制寄存器，低 4 位控制 T0 的工作方式，高 4 位控制 T1 的工作方式。

① M1、M0 是工作方式选择位，决定定时器的 4 种工作方式，如表 1-12 所示。

表 1-12 工作方式选择位

M1	M0	操 作 方 式	说 明
0	0	方式 0	13 位定时/计数器
0	1	方式 1	16 位定时/计数器
1	0	方式 2	8 位定时/计数器（定时常数自动装入）
1	1	方式 3	把 T0 分为两个 8 位计数器

② C/\overline{T} 是定时/计数方式选择位。 $C/\overline{T}=0$ 时为定时工作方式，在此方式下，计数脉冲来自单片机内部，计数脉冲频率为一个机器周期，机器周期的时间是固定的，所以就可根据计数值算出计数时间。当 TL0 初值为 6， $256-6=250$ ，当计数器计 250 个脉冲后，计数器溢出。当 $f_c=6\text{ MHz}$ 时，一个机器周期为 $2\text{ }\mu\text{s}$ ，因此 T0 溢出时，时间为 $500\text{ }\mu\text{s}$ ，T0 定时时间为 $500\text{ }\mu\text{s}$ 。

$C/\overline{T}=1$ 时为计数工作方式，在这种方式下，计数脉冲来自外部引脚（T0 对应 P3.4 脚，T1 对应 P3.5 脚）。当 T0 脚（或 T1 脚）发生从高电平到低电平的跳变时，计数器加 1。

③ GATE 是计数器工作方式控制位。当定时/计数器工作在计数方式时，由 GATE 设定计数器是否受外部控制。当 GATE=0 时，不受外部控制；当 GATE=1 时，计数器 T0 和 T1 分别受 P3.2 和 P3.3 脚上电平控制；当 P3.2（或 P3.3）脚为低电平时，计数器 T0（或 T1）开始计数；当 P3.2（或 P3.3）脚为高电平时，T0（或 T1）停止计数。

1.3.3 定时/计数器工作方式

C51 单片机的定时/计数器共有四种工作方式。工作在方式 0、方式 1 和方式 2 时，定时/计数器 0 和定时/计数器 1 的工作原理完全一样。

1. 方式 0（M1M0=00）

方式 0 是 13 位计数结构的工作方式，其计数器由 TH0 全部 8 位和 TL0 的低 5 位构成。TL0 高 3 位弃之不用。图 1-16 是定时/计数器 0 工作在方式 0 的逻辑结构。

当 $C/\overline{T}=0$ 时，多路转换开关接通振荡脉冲的 12 分频输出，13 位计数器以此作为计数脉冲，这时实现定时功能当多路换开关接通计数引脚（T0），计数脉冲由外部引入，当计数脉冲发生负跳变时，计数器加 1，这时实现计数功能。不管哪种功能，当 13 位计数发生溢出时，硬件自动把 13 位清零，同时硬件置位溢出标志位 TF0。

这里需要说明门控位（GATE）的用途，当 GATE=0 或输出的高电平与 $\overline{\text{INT}_0}$ 无关时，与门的输出仅受运行控制位 TR0 控制。如果 TR0=0，则与门输出为低电平，模拟开关断开，定时/计数器不工作；如果 TR0=1，则与门输出为高电平，则模拟开关闭合，定时/计数器工作。

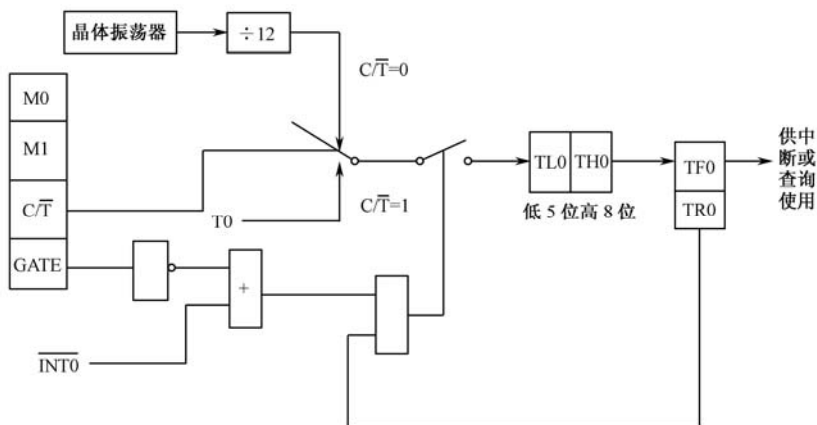


图 1-16 定时/计数器 0 工作在方式 0 逻辑结构

当 GATE=1 时, 只有 TR0 和 $\overline{\text{INT}}_0$ 同时为高电平, 定时/计数器才工作, 否则定时/计数器不工作。

- 定时和计数器的计数范围为 1~213;
- 计数计算公式: 计数值 = 213 - 计数初值;
- 定时范围为 1~213 机器周期;
- 定时计算公式: 定时时间 = (213 - 定时初值) × 机器周期;
- 如果晶振频率为 6 MHz, 初始值为 0, 则最大定时时间为 $2^{16} - (2^{13} - 1) \times 1/6 \text{ MHz} \times 12 = 2^{14} \mu\text{s}$ 。

2. 方式 1 (M1M0=01)

方式 1 是 16 位计数结构的工作方式，其计数器由 TH0 全部 8 位和 TL0 的全部 8 位构成。其逻辑电路和工作情况与方式 0 完全相同，所不同的只是计数器的位数。C51 单片机之所以设置几乎完全一样的方式 0 和方式 1，是出于为了满足 MCS-48 单片机兼容的要求，因为 MCS-48 单片机的定时/计数器是 13 位的计数结构。

- 定时和计数的计数范围为 1~216;
- 计数计算公式: 计数值 = 216 - 计数初值;
- 定时范围 1~216 机器周期;
- 定时计算公式: 定时时间 = (216 - 定时初值) × 机器周期;
- 如果晶振频率为 6 MHz, 初始值为 0, 则最大定时时间为 $2^{16} \times 1/6 \text{ MHz} \times 12 = 2^{14} \mu\text{s}$ 。

3. 方式 2 (M1M0=10)

方式 2 电路逻辑结构如图 1-17 所示。

由图 1-17 可以得出方式 2 具有以下特点:

- ① 8 位计数器;
- ② TL0 作计数器, TH0 作预置寄存器使用, 计数溢出时, TH0 中的计数初值自动装入 TL0, 即 TL0 是一个自动恢复初值的 8 位计数器;

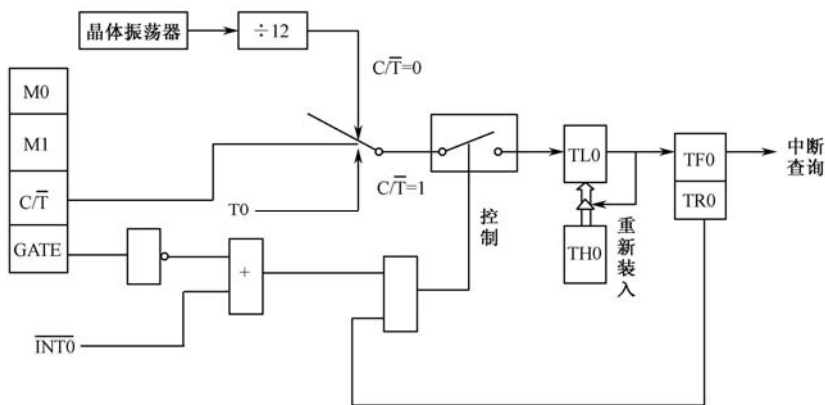


图 1-17 定时/计数器工作在方式 2 逻辑结构

③ 在使用时，要把计数初值同时装入 TL0 和 TH0 中；

④ 优点是提高定时精度，减少了程序的复杂程度。

- 定时和计数的计数范围：1~28；
- 计数计算公式：计数值 = 28 - 计数初值；
- 定时范围为 1~28 机器周期；
- 定时计算公式：定时时间 = (28 - 定时初值) × 机器周期。

4. 方式 3 (M1M0=11)

前面介绍的三种工作方式对两个定时/计数器而言，工作原理是完全一样的。但是在工作方式 3 下，两个定时/计数器工作原理却完全不同，因此要分开介绍。

(1) 工作方式 3 下的定时/计数器 0

在方式 3 下，定时/计数器 0 被拆为两个独立的 8 位的计数器 TL0 和 TH0，其中 TL0 既可以作为计数功能使用，又可以作为定时功能使用，共用定时/计数器 0 的运行控制位 TR0 和溢出标志位 TF0；对于 TH0，只能作定时器使用，由于定时/计数器 0 的运行控制位 TR0 和溢出标志位 TF0 已被 TL0 占用，因此，TH0 占用了定时/计数器 1 的运行控制位 TR1 和溢出标志位 TF1，即定时的启动和停止受 TR1 状态的控制，而计数溢出时则置位 TF1。

(2) 工作方式 3 下的定时/计数器 1

当定时/计数器 0 工作在方式 3 时，定时/计数器 1 只能工作在方式 0、方式 1 和方式 2。在这种情况下定时/计数器 1 只能作为波特率发生器使用，以确定串行通信的速率。作为波特率发生器使用时，只要设置好工作方式，便可自动运行。如果要停止工作，只需要把定时/计数器 1 设置在工作方式 3 就可以了，因为定时/计数器 1 不能工作在方式 3 下，如果把它设置为方式 3，它就会停止工作。

1.4 单片机的工作方式

C51 单片机中，80C51 的工作方式包括复位方式、程序执行方式、节电方式、低功耗方式以及 EPROM 编程和校验方式。不同的工作方式，表示单片机处于不同的状态。单片机工

作方式的多少是衡量单片机性能重要指标之一。

1.4.1 单片机的复位方式

复位操作是单片机的初始化操作，在单片机进入运行前和运行过程中程序出错或操作失误使单片机进入死锁状态时，都需要进行复位操作。复位是使 CPU 和系统中其他部件都处于一个确定的初始状态。复位操作后，07H 写入栈指针 SP，P0~P3 口均置 1，即允许输入，程序计数器 PC 和其他特殊功能寄存器 SFR 全部清 0。只要该脚保持高电平，单片机便循环复位。当 RST 端由高电平变为低电平后，程序将从 0000H 开始重新执行，单片机内部寄存器的值被初始化，如表 1-13 所示。除此之外，复位操作还使单片机的 ALE 和 PSEN 引脚信号在复位期间变为无效状态。

表 1-13 单片机复位后内部各寄存器状态

寄存器名	内 容	寄存器名	内 容
PC	0000H	TH0	00H
ACC	00H	TL0	00H
B	00H	TH1	00H
PSW	00H	TL1	00H
SP	07H	SBUF	不定
DPTR	0000H	TMOD	00H
P0~P3	FFH	SCON	00H
IP	×××0000B	PCON(HMOS)	0××××××B
IE	0××0000B	PCON(CHMOS)	0×××000B
TCON	00H		

单片机对复位信号有以下两个要求：

- ① RST 引脚是复位信号的输入端，复位信号为高电平 1 时有效；
- ② 复位信号有效持续时间不少于 2 个机器周期（24 个振荡脉冲）。若时钟频率为 6 MHz，则复位信号至少应持续 4 μs 以上，才可以使单片机复位。

在一个应用系统中，如果有多个单片机同时工作，在程序上有连接关系，系统复位时，应确保每一片单片机同时复位。复位操作不影响片内 RAM 单元的内容。复位操作会把 ALE 和 PSEN 变为无效状态，即 ALE=0，PSEN=1。

单片机复位方式包括上电自动复位和按键手动复位两种，图 1-18 是上电自动复位电路图，上电自动复位是通过电容充电来实现的。通过选择适当的电阻和电容的值，就能够使 RST 引脚上的高电平保持两个机器周期以上，在实现在上电的同时，完成单片机的复位。

按键手动复位包括按键电平复位和按键脉冲复位两种，如图 1-19 所示。

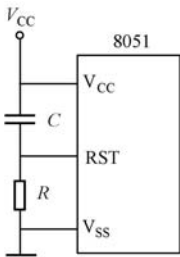


图 1-18 上电自动复位

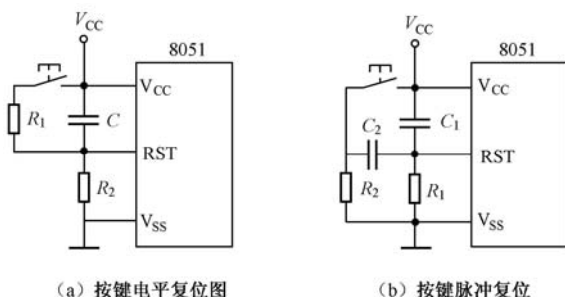


图 1-19 按键手动复位

图 1-19 所示的电路是通过引脚 RST 经电阻与电源相连接或利用 RC 微分电路产生的正脉冲来实现按键复位的，这两个电路同时也具备上电自动复位的功能。

1.4.2 程序执行方式

程序执行方式是单片机的基本工作方式，由于单片机复位后 PC=0000H，所以程序总是从地址 0000H 开始执行，此时单片机就根据指令的要求完成一系列的操作控制。考虑到单片机存储器结构的特殊性（0003H~0002BH 共 40 个单元，预留用于中断程序），在 0000H~0002H 中放一条无条件转移的指令，程序从指定的地址开始执行。

程序执行方式可分为连续执行和单步执行两种方式。

1. 连续执行方式

这是所有单片机都需要的一种方式。连续执行方式是从指定地址开始连续执行程序存储器 ROM 中存放的程序，每读一次程序，PC 自动加 1。单片机复位后，PC 值为 0000H，因此，单片机复位后立即转到 0000H 处执行程序。单片机按照程序事先编排的任务，自动连续地执行下去。

2. 单步执行方式

这是用户调试程序的一种工作方式。在单片机开发系统上有一个专用的单步按键，使程序的执行受控于外加脉冲（通常用按键产生），按一次按键，单片机就执行一条指令（仅执行一条），这样就可以逐条检查程序，以便发现问题并进行修改。

单步运行方式通常只在用户调试程序时使用，用于观察每条指令的执行情况。

单步执行方式可以通过 C51 单片机的外部中断功能实现。中断系统需要满足：从中断服务程序返回以后至少要执行一条指令后才能重新进入中断。将外加脉冲加到 INT0 输入，一般情况下为低电平。通过编程设置 INT0 信号低电平有效，因此，当没有脉冲出现时总是处于相应中断的状态。

在中断服务中要安排如下的指令：

```
JNB  P3.2, $      ; 程序不向下执行
JB   P3.2, $      ; 程序不向下执行
RESET                ; 返回主程序执行一条指令
```

因此，只有/INT0 引脚上出现一个正脉冲时，才能通过前两条指令，返回主程序并执行一条指令，这时/INT0 引脚上为低电平，重新进入中断，在第一条指令处等待正脉冲的出现，从而实现出现一个脉冲执行一条指令的单步操作。

1.4.3 节电工作方式

节电工作方式是一种低功耗的工作方式。C51 单片机中有 HMOS 和 CHMOS 两种工艺芯片，它们的节电运行方式不同，HMOS 单片机的节电工作方式只有掉电工作方式，CHMOS 单片机的节电工作方式有掉电工作方式和空闲工作方式两种。

HMOS 单片机的掉电保护：当 V_{CC} 端突然掉电时，单片机通过中断将必须保护的数据送入内部 RAM，备用电源端 V_{DD} 可以维持内部 RAM 中的数据不丢失。

CHMOS 单片机的节电方式：CHMOS 型单片机是一种低功耗器件，正常工作时电流为 11~22 mA，空闲状态时电流为 1.7~5 mA，掉电方式时电流为 5~50 μ A。因此，CHMOS 型单片机特别适用于低功耗应用场合，它的空闲方式和掉电方式都是由电源控制寄存器 PCON 中相应的位来控制。

单片机的节电工作方式是由其内部的电源控制寄存器 PCON 中的相关位来控制的。PCON 寄存器的控制格式如表 1-14 所示。

表 1-14 电源控制寄存器 PCON

位序	D7	D6	D5	D4	D3	D2	D1	D0
位符号	SMOD	—	—	—	GF1	GF0	PD	IDL

PCON 的各位定义如下：

- SMOD：串行口波特率倍率控制位，用于串行通信；
- GF1、GF0：通用标志位，描述中断是来自正常运行还是来自空闲方式，用户可通过指令设定它们的状态；
- PD：掉电方式控制位，PD=1 时，单片机进入掉电工作方式；
- IDL：空闲方式控制位，IDL=1 时，单片机进入空闲工作方式。

如同时将 PD 和 IDL 置 1，则进入掉电工作方式。PCON 寄存器的复位值为 0XXX0000，PCON.4~PCON.6 为保留位，用户不能对它们进行写操作。将 PD 和 IDL 置 0，则工作在正常运行状态。

1. 空闲工作方式

当程序将 PCON 的 IDL 位置 1（用指令 MOV PCON, #01H）后，系统就进入了空闲工作方式，其内部控制电路如图 1-20 所示。

空闲工作方式是在程序运行过程中，用户在 CPU 空闲时或不希望它执行程序时，进入的一种低功耗的待机工作方式。在此工作方式下，单片机的工作电流可降到正常工作方式时电流的 15% 左右。

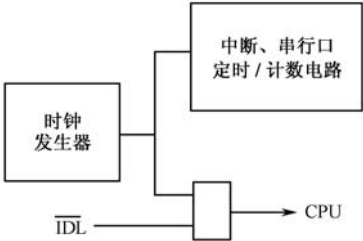


图 1-20 空闲工作方式内部控制电路

进入空闲工作方式后，提供给 CPU 的时钟信号被切断，CPU 停止工作，但时钟信号仍提供给 RAM、定时器、中断系统和串行口；同时堆栈指针 SP、程序计数器 PC、程序状态字 PSW、累加器 ACC 以及全部的通用寄存器都被冻结起来；I/O 引脚状态也保持不变。ALE 和 $\overline{\text{PSEN}}$ 保持逻辑高电平。单片机的消耗电流从 24 mA 降为 3.7 mA，这样就可以节省电源的消耗。

退出空闲方式的方法有两种，一种是中断退出，一种是按键复位退出。

任何的中断请求被响应都可以由硬件将 PCON.0 (IDL) 清 0，从而中止空闲工作方式。当执行完中断服务程序返回时，系统将从设置空闲工作方式指令的下一条指令继续开始执行程序。另外，PCON 寄存器中的 GF0 和 GF1 通用标志可用来指示中断是在正常情况下还是在空闲方式下发生的。例如，在执行设置空闲方式的指令前，先置位标志位 GF0（或 GF1），当空闲工作方式被中断中止时，在中断服务程序中可检测标志位 GF0（或 GF1），以判断出系统是在什么情况下发生中断的，如 GF0（或 GF1）为 1，则是在空闲方式下进入中断的。

另一种退出空闲方式的方法是按键复位，由于在空闲工作方式下振荡器仍然工作，因此复位仅需 2 个机器周期便可完成。而 RST 端的复位信号直接将 PCON.0 (IDL) 清 0，从而退出空闲状态，CPU 则从进入空闲方式的下一条指令开始重新执行程序。在内部系统复位开始，还可以有 2~3 个指令周期，在这一段时间里，系统硬件禁止访问内部 RAM 区，但允许访问 I/O 端口。通常，为了防止对端口的操作出现错误，在设置空闲工作方式指令的下一条指令中，不应该是对端口写或对外部 RAM 写指令。

2. 掉电工作方式

当 CPU 执行一条置 PCON.1 位 (PD) 为 1（用指令 MOV PCON, #02H）的指令后，系统立即进入掉电工作方式。

掉电的具体含义是指由于电源的故障使电源电压丢失，或工作电压低于要求的正常范围值。掉电将使单片机系统不能运行，若不采取保护措施，会丢失 RAM 和寄存器中的数据，因此单片机设置有掉电保护措施，进行掉电保护处理。

具体做法是：检测电路一旦发现掉电，立即先把程序运行过程中有用信息转存到 RAM，然后启用备用电源维持 RAM 供电。

在掉电工作方式下，单片机内部振荡器停止工作。由于没有振荡时钟，因此所有的功能部件都停止工作，时钟冻结，一切工作都停止。但内部 RAM 区和特殊功能寄存器的内容被保留，端口的输出状态值都保存在对应的 SFR 中，ALE 和 $\overline{\text{PSEN}}$ 都为低电平。这种工作方式下的电流可降到 15 μA 以下，最小可降到 0.6 μA 。

退出掉电方式的唯一方法是由硬件复位，复位时将所有的特殊功能寄存器的内容初始化，但不改变内部 RAM 区的数据。

在掉电工作方式下， V_{CC} 可以降到 2 V，但在进入掉电方式之前， V_{CC} 不能降低。而在准备退出掉电方式之前， V_{CC} 必须恢复正常的工作电压值，并维持一段时间（约 10 ms），使振荡器重新启动并稳定后方可退出掉电方式。实际上复位本身只需要 24 个振荡周期（2~4 μs ），但在进入掉电方式前， V_{CC} 不能掉下来，因此要有掉电监测电路。

空闲和掉电模式外部引脚状态如表 1-15 所示。

表 1-15 空闲和掉电模式外部引脚状态

模 式	程序存储器	ALE	$\overline{\text{PSEN}}$	P0	P1	P2	P3
空闲模式	内部	1	1	数据	数据	数据	数据
空闲模式	外部	1	1	浮空	数据	地址	数据
掉电模式	内部	0	0	数据	数据	数据	数据
掉电模式	外部	0	0	浮空	数据	数据	数据

1.4.4 EPROM编程和校验方式

对于片内程序存储器为 EPROM 型的单片机，如 8751 型单片机，需要一种对 EPROM 可以操作的工作方式，即用户可对片内的 EPROM 进行编程和校验。编程是指利用特殊手段对单片机片内部的 EPROM 进行写入的过程，主要操作是将原始程序和数据写入内部 EPROM 中。校验是对刚刚写入的程序代码进行读出校验的过程，主要操作是在向片内程序存储器 EPROM 写入信息时或写入信息后，将片内 EPROM 的内容读出进行校验，以保证写入信息的正确性。

8751H 和 8051 类似，不过 8751H 内部的 4 KB 程序存储器是 EPROM 型的，而 8051 是 ROM 型的。8751H 片内 EPROM 有编程、校验和保密编程等工作方式，在每种工作方式下，8751H 各引脚的输入电平是不相同的。

1. 内部EPROM的编程方式

在对单片机进行编程之前，其硬件电路连接如图 1-21 所示。

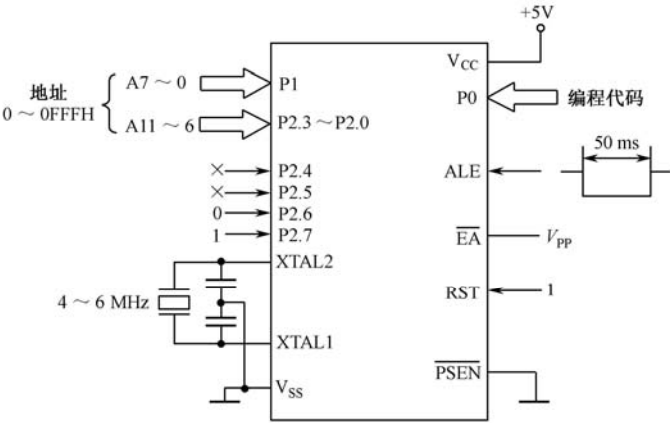


图 1-21 8751H 编程时电路连接

8751H 编程时电路连接电路说明如下：

- P2 口的 P2.0~P2.3 为 EPROM 的 4 KB 的高地址输入，P1 口为低 8 位地址；
- P2.4~P2.6 以及 $\overline{\text{PSEN}}$ 引脚为低电平；
- P0 口为编程数据输入；
- P2.7 以及 RST 为高电平，RST 的高电平可以为 2.5 V，其余的都以 TTL 的高电平为准；

- \overline{EA}/V_{pp} 引脚加 +12.5 V 的编程脉冲，这个电压要求稳定，不能大于 12.5 V，否则会损坏 EPROM；
- 在 \overline{EA}/V_{pp} 出现正脉冲之前，ALE/PROG 引脚上加 50 ms 的负脉冲，完成一次写入。

2. EPROM的校验方式

8751H 的 EPROM 的校验方式要求它的引脚按图 1-22 所示的电路进行连接。

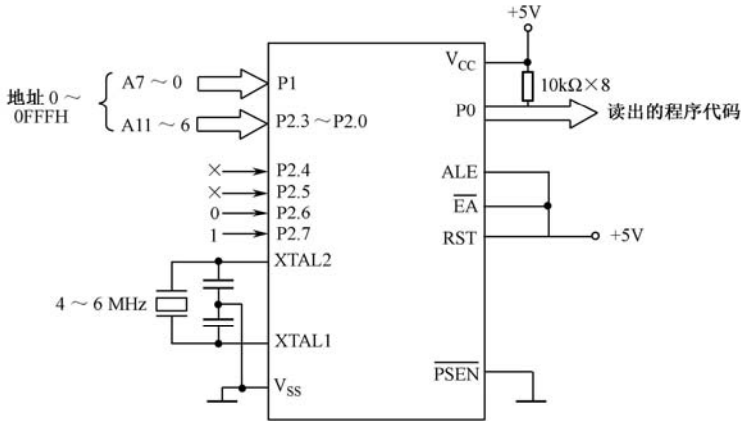


图 1-22 8751H 校验时的引脚连接

和编程方式类似，EPROM 校验也是在另一台主单片机控制下进行的。在校验时，主单片机把 12 位地址送入被校验的 8751H 的 P2 口和 P1 口，以选中并读出相应 EPROM 存储单元中内容，经 P0 口送给主单片机。主单片机把该读出代码和编程时写入的编程代码进行比较：若两者结果相同，则该单元编程正确；若结果不同，则应查明原因，重新进行编程，直到正确为止。

3. EPROM的保密方式

8751H 内部有一个保密位。它的硬件连接和图 1-6 所示电路的唯一区别在于 P2.6 应接逻辑高电平 1。

当将保密位写入后，就可以禁止任何外部对片内程序存储器进行读/写，内部存储区就不能被写入和读出校验，而且也不能执行外部程序存储器的程序，只有 EPROM 全部擦除时，保密位才能被一起擦除，这时才可以再次写入。

将保密位写入建立保险的过程，只需将 P2.6 加 TTL 高电平，而 P0 口、P1 口和 P2 口的 P2.0 ~ P2.3 状态任意，加上编程脉冲后就可以使保密位写入。

1.5 C51 单片机的指令系统

单片机的功能是从外部接收信息，经过 CPU 加工、处理后，再把结果送出来，因此在 CPU 内部需要一整套特定功能的操作指令，一个单片机所需执行指令的集合即为单片机的指令系统。单片机可以使用机器语言、汇编语言和高级语言，但不管使用何种语言，最终还是要“翻译”成为机器码，单片机才能执行。

1.5.1 计算机语言

目前单片机种类繁多，品种数不胜数，值得注意的是不同单片机的指令系统不一定相同或不完全相同。计算机语言通常分为机器语言、汇编语言和高级语言三类，但不管是使用机器语言、汇编语言还是高级语言，都是使用指令编写程序的。

1. 机器语言

机器语言是用二进制代码表示的计算机能直接识别和执行的一种机器指令的集合，它是计算机的设计者通过计算机的硬件结构赋予计算机的操作功能。用机器语言编写程序，首先应熟记所用计算机的全部指令代码和代码的含义，编出的程序全是些 0 和 1 的指令代码，直观性差，容易出错。机器语言具有灵活、直接执行和速度快等特点。

2. 汇编语言

为了克服机器语言难读、难编、难记和易出错的缺点，人们用与代码指令实际含义相近的英文缩写词、字母和数字等符号来取代指令代码（例如，用 **ADD** 表示运算符“+”的机器代码），于是就产生了汇编语言。汇编语言是一种用助记符表示的、仍然面向机器的计算机语言，汇编语言也称为符号语言。在指令的表达式上不直接使用二进制机器码，最常用的是采用十六进制的形式。汇编语言的特点是用符号代替了机器指令代码，而且助记符与指令代码一一对应，基本保留了机器语言的灵活性。使用汇编语言能面向机器并较好地发挥机器的特性，得到质量较高的程序。

汇编语言中由于使用了助记符号，用汇编语言编制的程序送入计算机，计算机不能像用机器语言编写的程序一样直接识别和执行，必须通过预先存放在计算机的汇编语言程序的加工和翻译，才能变成能够被计算机识别和处理的二进制代码程序。用汇编语言等非机器语言编写的符号程序称为源程序，运行时汇编语言程序将源程序翻译成目标程序。目标程序是机器语言程序，它一经被安置在内存的预定位置上，就能被计算机的 CPU 处理和执行。

汇编语言和机器指令一样，是硬件操作的控制信息，因而仍然是面向机器的语言，使用起来还是比较烦琐费时的，通用性也比较差。汇编语言是低级语言，但是，用汇编语言编制的系统软件和过程控制软件，其目标程序占用内存空间少，运行速度快，有着高级语言不可替代的优点。

3. 高级语言

不论是机器语言还是汇编语言，它们都是面向硬件的具体操作的，语言对机器的过分依赖，要求使用者必须对硬件结构及其工作原理都十分熟悉。计算机事业的发展，促使人们去寻求一些与人类自然语言相接近且能为计算机所接受的、语意确定的、规则明确的、自然直观的和通用易学的计算机语言。这种与自然语言相近并为计算机所接受和执行的计算机语言称为高级语言，高级语言是面向用户的语言。无论何种机型的计算机，只要配备上相应的高级语言的编译或解释程序，用该高级语言编写的程序就可以通用。

目前被广泛使用的高级语言有 **BASIC** 语言、**PASCAL** 语言、**C** 语言、**COBOL** 语言、

FORTRAN 语言、LOGO 语言以及 VC、VB 等。

计算机并不能直接地接收和执行用高级语言编写的源程序。程序在输入计算机时，经过“翻译程序”翻译成机器语言形式的目标程序，才能被计算机识别和执行。这种“翻译”通常有编译和解释两种方式。

(1) 编译方式

事先编制一个称为编译程序的机器语言程序，作为系统软件存放在计算机内，当用户由高级语言编写的源程序输入计算机后，编译程序便把源程序翻译成用机器语言表示的、与之等价的目标程序，然后计算机再执行该目标程序，以完成源程序要处理的运算并取得结果。PASCAL、FORTRAN、COBOL 等高级语言以执行编译方式为主。

(2) 解释方式

源程序进入计算机时，解释程序边扫描边解释，逐句输入逐句翻译，计算机一句句执行，并不产生目标程序。BASIC 语言则以执行解释方式为主。

每一种高级（程序设计）语言，都有自己人为规定的专用符号、英文单词、语法规则和语句结构（书写格式）。高级语言与自然语言（英语）更接近，而与硬件功能相分离（彻底脱离了具体的指令系统），便于掌握和使用。高级语言的通用性强，兼容性好，便于移植。

每种单片机都有自己独特的指令系统，那么指令系统是开发和生产厂商定义的，如要使用其单片机，用户就必须理解和遵循这些指令标准，要掌握某种（类）单片机，指令系统的学习是必须的。

1.5.2 C51 单片机的寻址方式

寻址方式是指定操作数所在单元的方法。寻址的“地址”为操作数所在单元的地址，绝大部分指令执行时都需要用到操作数，告诉 CPU 操作数所在的地址单元，从哪里可取得相应的操作数，这便是寻址之意。

例如指令：

MOV P1, #03AH

这条指令中，第一个词 MOV 是命令动词，也就是决定做什么事情的，“MOV”是“MOVE”这个单词的简写形式，意为“传送”，这就是指令，规定做什么事情，数据传递必须要有一个“源”，也就是要送什么数，必须要有一个“目的”，也就是这个数要送到什么地方去，显然在上面那条指令中，要送的数（源）就是 03AH，而要送达的地方（目的地）就是 P1 这个寄存器。

学习汇编语言程序设计，要先了解 CPU 的各种寻址法，才能有效地掌握各个命令的用途。C51 单片机的寻址方式很多，使用起来也相当方便，功能强大，灵活性强。

C51 单片机指令系统共有 7 种寻址方式，包括立即寻址、直接寻址、寄存器寻址、寄存器间接寻址、变址寻址、相对寻址和位寻址。应掌握指令的 7 种寻址方式的作用以及不同寻址方式所查询的存储空间及范围，对于常用的指令，能够给出指令的寻址方式。

下面分别讨论几种寻址方式的原理。

1. 立即寻址

指令中操作数是二进制数或十进制数，出现在指令中，用“#”做单元地址形式的前缀，寻址空间为程序存储器。

在这种寻址方式中，指令多是双字节的，一般第一个字节是操作码，第二个字节是操作数。该操作数直接参与操作，所以又称为立即数，用“#”表示。立即数就是存放在程序存储器中的常数，即操作数（立即数）是包含在指令字节中的。

例如指令：

MOV A, #1BH

这条指令的指令代码是双字节指令，其功能是把立即数 1BH 送入累加器 A 中。该指令的执行过程如图 1-23 所示。

MOV DPTR, #8200H

把立即数的高 8 位（即 82H）送入 DPH 寄存器，把立即数的低 8 位（即 00H）送入 DPL 寄存器。

这里也特别说明一下：在 80C51 单片机的指令系统中，仅有一条指令的操作数是 16 位的立即数，其功能是向地址指针 DPTR 传送 16 位的地址，即把立即数的高 8 位送入 DPH 寄存器，低 8 位送入 DPL 寄存器。

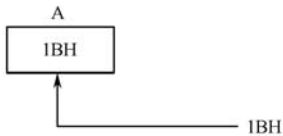


图 1-23 立即数寻址示意图

2. 直接寻址

直接寻址是直接在操作码后面的字节给出实际操作数地址的方式，在这种寻址方式中，操作数给出的是参加运算的操作数的地址，而不是操作数。寻址空间包括内部 RAM 的低 128 字节和特殊功能寄存器 SFR。

例如指令：

MOV A, 68H

这条指令的意义是把内部 RAM 中的 68H 单元中的数据内容传送到累加器 A 中。直接寻址方式只能使用 8 位二进制地址，因此这种寻址方式仅限于内部 RAM 进行寻址。低 128 单元在指令中直接以单元地址的形式给出。对于特殊功能寄存器可以使用其直接地址进行访问，也可以以它们的符号形式给出。直接寻址是唯一能访问特殊功能寄存器的寻址方式，特殊功能寄存器只能用直接寻址方式访问，而无其他方法。

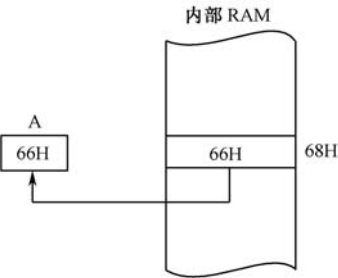


图 1-24 直接寻址示意图

假设内部 RAM 68H 单元的内容是 66H，那么指令 MOV A, 68H 的执行过程如图 1-24 所示。

在 80C51 单片机中，直接寻址只能用来表示特殊功能寄存器、内部数据存储器以及位地址空间，具体的说就是：

① 内部数据存储器 RAM 低 128 单元，在指令中是以直接单元地址形式给出。低 128 单元的地址是 00H~7FH，在指令中直接以单元地址形式给出，意思就是这 0~127 共 128 位的任何一位，例如 0 位是以 00H 这个单元地址形式给出，1 位是以 01H 单元地址给出，127 位就是以 7FH 形式给出。

- ② 位寻址区包含 20H~2FH 地址单元。
- ③ 特殊功能寄存器。专用寄存器除了可以以单元地址形式给出外，还可以以寄存器符号形式给出。

3. 寄存器寻址

寄存器寻址用于对选定的 8 个工作寄存器 R0~R7 进行操作，即操作数在寄存器中，因此指定了寄存器就得到了操作数，寄存器寻址的指令中以寄存器的符号来表示寄存器。

例如指令：

```
MOV R1, A
```

指令的操作是把累加器 A 中的数据传送到寄存器 R1 中，其操作数存放在累加器 A 中，所以寻址方式为寄存器寻址。如果程序状态寄存器 PSW 的 RS1RS0=01（选中第二组工作寄存器，对应地址为 08H~0FH），设累加器 A 的内容为 20H，则执行 MOV R1, A 指令后，内部 RAM 09H 单元的值就变为 20H，如图 1-25 所示。

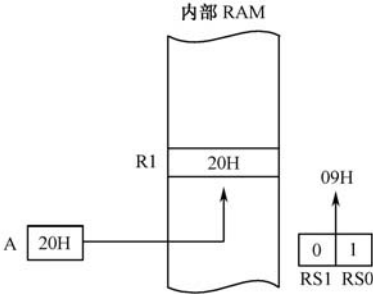


图 1-25 寄存器寻址示意图

把所用的工作寄存器组中的 R3 的内容送到累加器 A 中，操作数就在 R0 中。

```
INC R3 ; 把寄存器 R3 中的内容加 1
```

寄存器寻址的寻址范围是：

- ① 4 个工作寄存器组共有 32 个通用寄存器。但在指令中只能使用当前寄存器组，因此在使用前常需要通过对 PSW 中的 RS1、RS0 位的状态设置，来进行对当前工作寄存器组的选择。
- ② 部分专用寄存器。例如，累加器 A、通用寄存器 B、地址寄存器 DPTR 和进位位 Cy。

4. 寄存器间接寻址

寄存器寻址方式，寄存器中存放的是操作数，而寄存器间接寻址方式，寄存器中存放的则为操作数的地址，即操作数是通过寄存器间接得到的，因此称为寄存器间接寻址。“@”表示前缀。

C51 单片机规定工作寄存器的 R0、R1 作为间接寻址寄存器，用于寻址内部或外部数据存储器的 256 个单元。

例如指令：

MOV R0, #30H ; 将值 30H 加载到 R0 中
MOV A, @R0

上述指令把内部 RAM 地址 30H 内的值放到累加器 A 中，如果 R0=#56H，那么是将 56H 单元中的数据送到累加器 A 中，该指令的执行过程如图 1-26 所示。

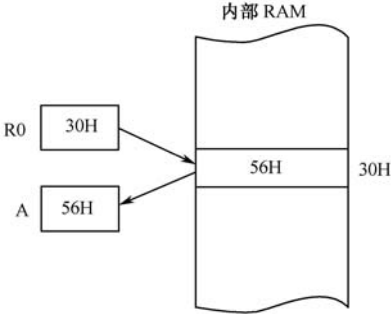


图 1-26 寄存器间接寻址示意图

MOVX A, @R0 ; 把外部 RAM 地址 30H 内的值放到累加器 A 中

如果用 DPTR 作为间接寻址寄存器，由于 DPTR 是一个 16 位的寄存器，所以它的寻址范围就是 $2^{16} = 65536$ 。因为用 DPTR 作为间接寻址寄存器的寻址空间是 64 KB，所以访问片外数据存储单元时，通常使用 DPTR 作为间接寻址寄存器。

例如指令：

MOV DPTR, #1234H ; 将 DPTR 值设为 1234 H（16 位）

MOVX A, @DPTR ; 将外部 RAM 或 I/O 地址 1234 H 内的值放到累加器 A 中
在执行 PUSH（压栈）和 POP（出栈）指令时，采用堆栈指针 SP 作寄存器间接寻址。

例如指令：

PUSH 30H ; 把内部 RAM 地址 30H 内的值放到堆栈中

堆栈是由 SP 寄存器指定的，如果在执行上面这条命令前，SP 为 60 H，命令执行后会
把内部 RAM 地址 30H 内的值放到 RAM 的 61 H 内。

寄存器间接寻址范围为：

- ① 内部 RAM 低 128 单元。对内部 RAM 低 128 单元的间接寻址，应使用 R0 或 R1 作为间接寻址寄存器，其通用形式为 “@Ri”（i=0 或 1）。
- ② 外部 RAM 64 KB。对外部 RAM 64 KB 的间接寻址，应使用 @DPTR 作为间接寻址寄存器，其形式为 “@DPTR”。
- ③ 堆栈操作指令（PUSH 和 POP）也应算做寄存器间接寻址，即以堆栈指针 SP 作为间接寻址寄存器的间接寻址方式。
- ④ 寄存器间接寻址方式不可以访问特殊功能寄存器。

5. 变址寻址

变址寻址方式是 C51 单片机所独有的一种寻址方式。变址寻址是以 DPTR 或 PC 作为基址寄存器，以累加器 A 作为变址寄存器，将两寄存器的内容相加形成 16 位地址形成操作数的实际地址。这种方式常用于访问程序存储器 ROM 中的数据表格，即查表操作。变址寻址

只能读出程序内存入的值，而不能写入，即变址寻址这种寻址方式只能对程序存储器进行寻址，或者说它是专门针对程序存储器的寻址方式。

例如指令：

```
MOVC A, @A+DPTR
```

这条指令的功能是把 DPTR 和 A 的内容相加，再把所得到的程序存储器地址单元的内容送 A。假若指令执行前累加器 A=02H，DPTR=0300H，则这条指令变址寻址形成的操作数地址就是 02H+0300H=0302H。如果 3F75H 单元中的内容是 7FH，则执行这条指令后，累加器 A 中的内容就是 7FH。该指令的执行过程如图 1-27 所示。

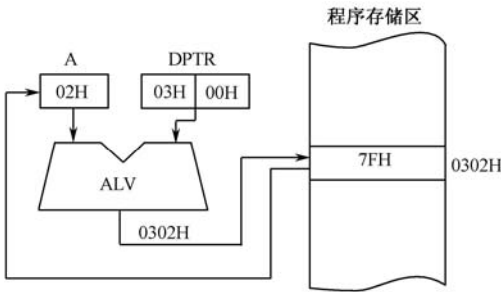


图 1-27 变址寻址示意图

变址寻址的指令只有三条，分别如下。

(1) JMP @A+DPTR

这是一条无条件转移的指令，这条指令的意思就是 DPTR 加上累加器 A 的内容作为一个 16 位的地址，执行 JMP 这条指令时，程序就转移到 A+DPTR 指定的地址去执行。

(2) MOVC A, @A+DPTR

这条指令用于查表操作。DPTR 是一个 16 位的数据指针寄存器，可以给数据指针 DPTR 进行赋值，通过赋值操作，寻址范围达到 64 KB。

(3) MOVC A, @A+PC

这条指令用于查表操作。PC 是程序指针，PC 的值是随程序的执行情况自动改变的，不可以随便的给 PC 赋值。这条指令的意思是将 PC 的值与累加器 A 的值相加作为一个地址，而 PC 是固定的，累加器 A 是一个 8 位的寄存器，它的寻址范围是 256 个地址单元。

6. 位寻址

C51 单片机有位处理功能，可以对数据位进行操作，因此就有相应的位寻址方式。所谓位寻址，就是对内部 RAM 或可进行位寻址的特殊功能寄存器 SFR 内的某个位，直接置位为 1 或复位为 0。

例如指令：

```
SETB 3DH
```

这条指令执行的操作是将内部 RAM 位寻址区中的 3DH 位置 1。假设内部 RAM 27H 单元的内容是 00H，执行 SETB 3DH 后，由于 3DH 对应着内部 RAM 27H 的第 5 位，因此该位变为 1，也就是 27H 单元的内容变为 20H，该指令的执行过程如图 1-28 所示。

位寻址的范围为：

① 学习 C51 单片机的存储器结构时，应当注意单片机的内部数据存储器 RAM 的低 128 单元中有一个区域叫做位寻址区，它的单元地址是 20H~2FH，共有 16 个单元，一个单元是 8 位，所以位寻址区共有 128 位。这 128 位都单独有一个位地址，其位地址是 00H~7FH。

② 对于专用寄存器位寻址而言，不是所有的专用寄存器都可以位寻址的，一般来说，地址单元可以被 8 整除的专用寄存器，通常都可以进行位寻址，当然并不是全部，大家在应用当中需要注意。

在前面的学习中，00H、01H、…、7FH 等所表示的都是一个字节（或者叫单元地址），而在这里，这些数据都变成了位地址。在指令中，或者在程序中如何来区分它是一个单元地址还是一个位地址呢？其实，区分这些数据是位地址还是单元地址，都有相应的指令形式的。

7. 相对寻址

相对寻址方式是为了程序的相对转移而设计的，以指令所在单元地址（PC 内容），加上给出的地址偏移量作为真正有效的操作数所存放的转移地址，从而实现程序的转移。即

目的地址 = 转移指令地址 + 转移指令字接数 + 偏移量

值得注意的是，偏移量是有正负号之分的，偏移量的取值范围是当前 PC 值，即 -128~+127 之间。

例如指令：

SJMP 54H

该指令执行的操作是将 PC 当前的内容与 54H 相加，结果再送回 PC 中，成为下一条将要执行指令的地址。假设的机器码 80H 和 54H 存放在 2000H 处，当执行到该指令时，先从 2000H 和 2001H 单元取出指令，PC 自动变为 2002H；再把 PC 的内容与操作数 54H 相加，形成目标地址 2056H，再送回 PC，使得程序跳转到 2056H 单元继续执行。该指令的执行过程如图 1-29 所示。

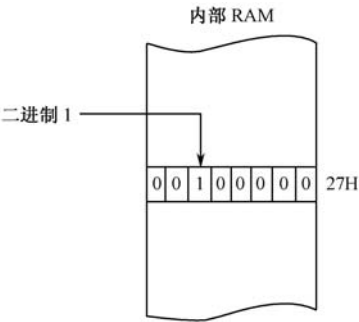


图 1-28 位寻址示意图

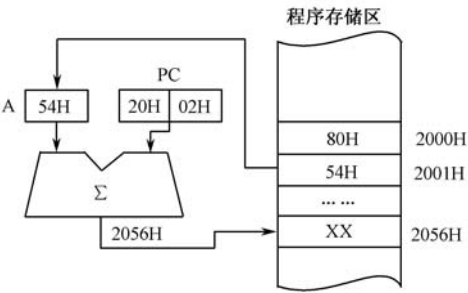


图 1-29 相对寻址示意图

下面以程序状态字 PSW 来说明专用寄存器的位寻址表示方法。程序状态字 PSW 的寄存器如表 1-16 所示。

表 1-16 程序状态字 PSW 的寄存器

位序	D7	D6	D5	D4	D3	D2	D1	D0
位符号	CY	AC	F0	RS1	RS0	OV	—	P

专用寄存器的位寻址表示方法有如下四种。

- ① 直接使用位地址表示：PSW 的第 5 位地址是 D5，所以可以表示为 D5H。
例如指令：MOV C, D5H。
 - ② 位名称表示：表示该位的名称，例如 PSW 的位 5 是 F0，所以可以用 F0 表示。
例如指令：MOV C, F0。
 - ③ 单元（字节）地址加位表示：D0H 单元位 5，表示为 D0H.5。
例如指令：MOV C, D0H.5。
 - ④ 专用寄存器符号加位表示：例如 PSW.5。
例如指令：MOV C, PSW.5。
- 这四种方法实现的功能都是相同的，只是表述的方式不同而已。

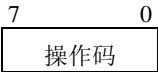
1.5.3 C51 单片机的指令系统

C51 单片机指令系统有如下特点：

- 指令执行速度快；
- 指令短，约有一半的指令为单字节指令；
- 用一条指令即可实现 2 个单字节的相乘或相除；
- 具有丰富的位操作指令；
- 可直接用传送指令实现端口的输入/输出操作。

C51 单片机指令按照不同指令翻译成机器码后字节数也不一定相同，按照机器码个数，指令可以分为以下三种：

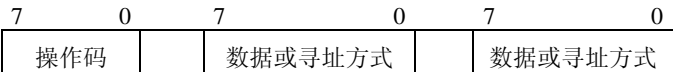
- ① 单字节指令，即



- ② 双字节指令，即



- ③ 三字节指令，即



C51 单片机指令系统包括 49 条单字节指令、46 条双字节指令和 16 条三字节指令。

由操作码助记符字段和操作数字段组成的汇编语言指令。指令格式如下：

标号：操作码 [操作数 1] [,操作数 2] [,操作数 3] [:注释]

标号：是根据编程需要给指令设定的符号地址，可有可无；标号必须以字母开始，由

1~8 个字母或数字组成，加在指令之前，并以“:”结尾。用户定义的标号不能和汇编保留符号（包括指令操作码助记符以及寄存器名等）重复。标号的值是它后面的指令存储地址。

操作码：由 2~5 个英文字母所组成，如 JB、MOV、CJNE、LCALL 等。

操作数：以一个或几个空格和操作码隔开，根据指令功能的不同，操作数可以有 1、2、3 个或没有（如 NOP）。操作数之间以“,”分开。

注释：以“;”和操作数分开。如果一行写不下，可以另起一行，但都必须以“;”开始。注释的作用是用户对某一条指令或某一段程序的功能说明，在指令中不起作用。指令汇编后，注释没有被汇编成机器码。

C51 单片机采用复杂指令系统，共有 33 种功能，42 种操作码助记符，支持直接寻址、寄存器寻址、间接寻址、立即数寻址、变址寻址、相对寻址、位寻址 7 种寻址方式。不同指令操作码助记符与不同寻址方式之间的组合构成了 C51 单片机的指令系统，共计 111 条指令，可分为 5 类：

- 数据传送类指令（共 29 条）；
- 算数运算类指令（共 24 条）；
- 逻辑运算及移位类指令（共 24 条）；
- 控制转移类指令（共 17 条）；
- 布尔变量操作类指令（共 17 条）。

1.5.4 指令系统中的符号说明

在介绍指令系统前，先了解一些特殊符号的意义，如表 1-17 所示。

表 1-17 特殊符号的意义

符 号	符号表示的意义
Rn	当前选定寄存器组的 8 个工作寄存器 R0~R7
Ri	当前选中的寄存器区中可作为间接寻址的地址指针 R0 和 R1
direct	内部数据存储单元的 8 位地址，也可以是内部 RAM 区的某一单元或某一专用功能寄存器的地址
#data	指令中的 8 位常数（立即数），即 00H~FFH
#data16	指令中的 16 位常数（立即数），即 0000H~FFFFH
addr16	用于 LCALL 和 LJMP 指令中的 16 位目的地地址，目的地址的空间为 64 KB 程序存储器地址
addr11	用于 ACALL 和 AJMP 指令中的 11 位目的地地址，目的地址必须放在与下条指令第一个字节同一个 2 KB 程序存储器空间之中
Rel	带符号的 8 位偏移量（-128~+127），用于所有的条件转移和 SJMP 等指令中
Bit	内部 RAM 和特殊功能寄存器的直接寻址位
@	间接寄存器寻址或基址寄存器的前缀
/	为操作的前缀，声明对该位操作数取反
DPTR	数据指针
A	累加器
B	累加器 B，用于乘法和除法指令中
C	进位标志位

符 号	符号表示的意义
(x)	某地址单元中的内容
((X))	将 X 地址单元中的内容作为地址, 该地址单元中的内容
←	将“←”后面的内容传送到前面去

1. 数据传送指令

在 C51 单片机汇编语言程序设计中, 使用最频繁的指令是数据传送指令, 数据传送指令一般的操作是把源操作数传送到目的操作数, 指令执行完成后, 源操作数不变, 目的操作数等于源操作数, 包括内部 RAM、寄存器、外部 RAM 以及程序存储器之间的数据传送。

C51 单片机片内数据传送途径如图 1-30 所示。

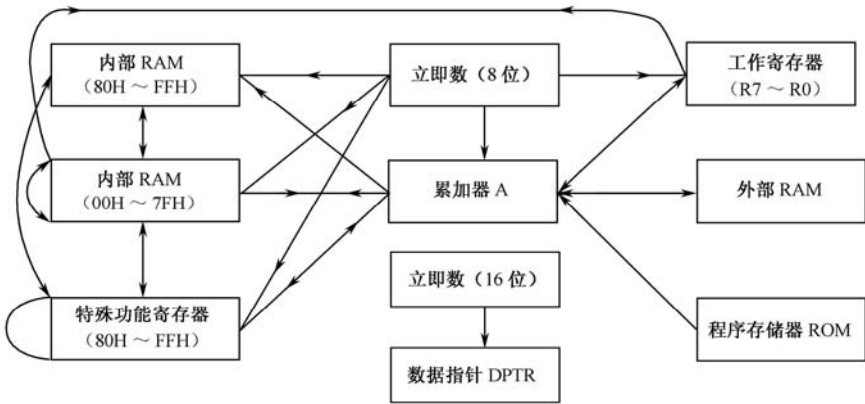


图 1-30 C51 单片机片内数据传送途径

数据传送指令一般不影响程序状态字寄存器 PSW 中的标志位, 只有当数据传送到累加器 A 时, PSW 中的奇偶标志位 P 才会改变, 原因是奇偶标志位 P 总是体现累加器 A 中 1 的个数的奇偶性。当目的操作数为累加器 A 时, 数据传送指令会影响 Z 标志, 比如当累加器 A 为 0 时, Z (零) 标志置 1; 反之 Z 标志清 0。

(1) 内部 RAM、特殊功能寄存器之间的数据传送

内部 RAM、特殊功能寄存器之间的数据传送用“MOV”作为指令操作码助记符。

① 以累加器 A 作为目的操作数。指令的作用是把源操作数指向的内容送到累加器 A。源操作数可采用直接、立即数、寄存器和寄存器间接寻址方式。例如指令：

```
MOV A, data      ; 直接单元地址中的内容送到累加器 A
MOV A, #data     ; 立即数送到累加器 A 中
MOV A, R3        ; R3 中的内容送到累加器 A 中
MOV A, @R1       ; R1 内容指向的地址单元中的内容送到累加器 A
```

② 以寄存器 Rn 为目的操作数的指令。指令的功能是把源操作数指定的内容送到所选定的工作寄存器 Rn 中。源操作数可采用直接、立即和寄存器寻址方式。例如指令：

```
MOV Rn, data     ; 直接寻址单元中的内容送到寄存器 Rn 中
MOV Rn, #data    ; 立即数直接送到寄存器 Rn 中
```

MOV Rn, A ; 累加器 A 中的内容送到寄存器 Rn 中

③ 以直接地址 direct 为目的操作数的指令。指令的功能是把源操作数指定的内容送到由直接地址所选定的片内 RAM 中。源操作数可采用直接、立即、寄存器和寄存器间接 4 种寻址方式。例如指令：

MOV direct, A

MOV direct, data, data ;(data), (data) 直接地址单元中的内容送到直接地址单元

MOV direct, #data ; 立即数送到直接地址单元

MOV direct, A ; 累加器 A 中的内容送到直接地址单元

MOV direct, Rn ; 寄存器 Rn 中的内容送到直接地址单元

MOV direct, @Ri ; 寄存器 Ri 中的指定的地址单元中数据送到直接地址单元

④ 以间接地址为目的操作数的指令。指令的功能是把源操作数指定的内容送到以 Ri 中的内容为地址的片内 RAM 中。源操作数可采用直接、立即和寄存器 3 种寻址方式。例如指令：

MOV @Ri, direct ; 直接地址单元中的内容送到以 Ri 中的内容为地址的 RAM 单元

MOV @Ri, #data ; 立即数送到以 Ri 中的内容为地址的 RAM 单元

MOV @Ri, A ; 累加器 A 中的内容送到以 Ri 中的内容为地址的 RAM 单元

⑤ 16 位数据传送指令。指令的功能是把 16 位常数送入数据指针寄存器。例如指令：

MOV DPTR, #data16 ; 16 位常数的高 8 位送到 DPH, 低 8 位送到 DPL

(2) 外部 RAM 及 I/O 口与累加器 A 之间的数据传送

在 C51 单片机中, 外部 RAM 及扩展 I/O 口的读/写操作指令、操作时序完全相同, 只能通过累加器 A 存取外部 RAM 和扩展 I/O 口, 这类指令用“MOVX”作为指令操作码助记符, 其中“X”的含义是“eXternal”(外部)。源操作码可以采用寄存器寻址方式。例如指令：

MOVX @DPTR, A ; 累加器中的内容送到数据指针指向片外 RAM 地址中

MOVX A, @DPTR ; 数据指针指向片外 RAM 地址中的内容送到累加器 A 中

MOVX A, @Ri ; 寄存器 Ri 指向片外 RAM 地址中的内容送到累加器 A 中

MOVX @Ri, A ; 累加器中的内容送到寄存器 Ri 指向片外 RAM 地址中

(3) 累加器 A 与程序存储器 ROM 之间的数据传送指令

C51 单片机提供查表指令, 用来取出存放在程序存储器中的表格数据, 这两条指令的操作码助记符为“MOVC”, 其中“C”的含义是“code(代码)”, 表示操作对象是程序存储器。

查表指令的功能是对存放于程序存储器中的数据表格进行查找传送, 源操作数可采用变址寻址方式。例如指令：

MOVC A, @A+DPTR ; 表格地址单元中的内容送到累加器 A 中

MOVC A, @A+PC ; 表格地址单元中的内容送到累加器 A 中

(4) 堆栈操作指令

堆栈操作也是计算机系统基本操作之一。所谓堆栈是在片内 RAM 中按“先进后出, 后进先出”原则设置的专用存储区。数据的进栈出栈由指针 SP 统一管理, 使用时应先设堆栈

指针 SP，堆栈指针 SP 默认值为 07H。设置堆栈操作的目的是为了迅速保护断点和现场，以便在子程序或中断服务子程序运行结束后，能正确返回主程序。

这类指令只有两条，入栈操作指令（PUSH）和出栈操作指令（POP），指令结果不影响程序状态字寄存器 PSW 标志。堆栈操作必须是字节操作，且只能直接寻址。将累加器 A 入栈、出栈指令可以写成 PUSH/POP ACC 或 PUSH/POP 0E0H，而不能写成 PUSH/POP A。

PUSH 是进栈（或称为压入操作）指令，指令执行过程如图 1-31 所示。

指令格式：

PUSH direct ；将 SP 加 1，然后将源地址单元中的数传送到 SP 所指示的单元中去

POP 是出栈指令。指令执行过程如图 1-32 所示。

指令格式：

POP direct ；将 SP 所指示的单元中的数传送到 direct 地址单元中，然后 SP 减 1

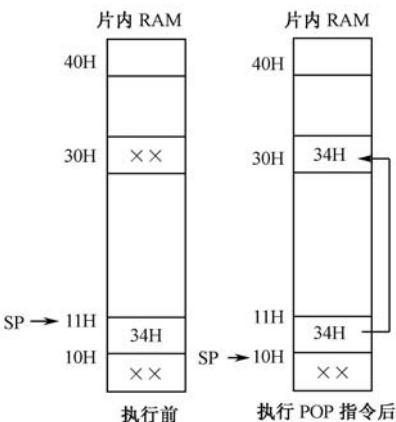
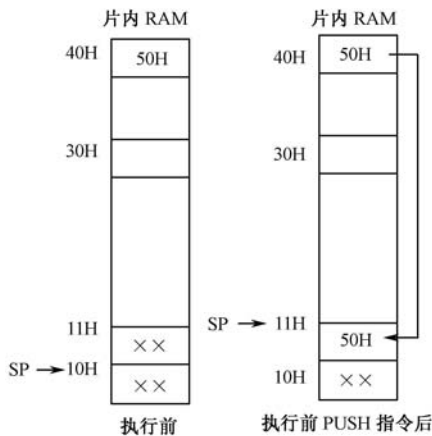


图 1-31 指令 PUSH 操作示意图

图 1-32 指令 POP 操作示意图

(5) 交换指令

C51 单片机提供了四条字节交换指令和两条半字节交换指令。交换指令也属于数据传送指令，不过交换后，源操作数与目的操作数内容相互对调。交换指令包括字节交换指令和半字节交换指令。

① 字节交换。例如

XCH A, Rn ；累加器与工作寄存器 Rn 中的内容互换

XCH A, direct ；累加器与直接地址单元中的内容互换

XCH A, @Ri ；累加器与工作寄存器 Ri 所指的存储单元中的内容低半字节互换

② 半字节交换。例如

XCHD A, @Ri ；累加器与工作寄存器 Ri 所指的存储单元中的内容低半字节互换

SWAP A ；累加器中的内容高低半字节互换

2. 算术运算指令

C51 单片机提供了 24 条算术运算指令，算术运算主要是执行加法、减法、乘法、除法四则运算。C51 单片机的算术逻辑单元 ALU 仅能对 8 位无符号整数进行运算，但利用进位

标志 C，则可进行多字节无符号整数的运算。同时利用溢出标志，还可以对带符号数进行补码运算。除加、减 1 指令外，其余的算术运算指令执行后都会影响程序状态字寄存器 PWS 中相应的标志位。

C51 提供了丰富的算术运算指令，如加法运算、减法运算、加 1 指令、减 1 指令以及乘法、除法指令等。

(1) 加法指令

加法指令的作用是把立即数、直接地址、工作寄存器及间接地址内容与累加器 A 的内容相加，运算结果存在 A 中。

加法指令操作码助记符、指令格式以及机器码如表 1-18 所示。

表 1-18 C51 单片机加法指令

指令名称	助记符格式	机器码 (B)	相应操作	指令说明	机器周期
不带进位 加法指令	ADD A, Rn	00101rrr	累加器 A 中的内容与工作寄存器 Rn 中的内容相加，结果存在 A 中	n=0~7 rrr=000~111	1
	ADD A, direct	00100101	累加器 A 中的内容与直接地址单元中的内容相加，结果存在 A 中		1
	ADD A, @Ri	0010011i	累加器 A 中的内容与工作寄存器 Ri 所指向地址单元中的内容相加，结果存在 A 中	i=0, 1	1
	ADD A, #data	00100100 data	累加器 A 中的内容与立即数#data 相加，结果存在 A 中		1
带进位 加法指令	ADDC A, Rn	00111rrr	$A \leftarrow A + Rn + Cy$ 累加器 A 中的内容与工作寄存器 Rn 中的内容、连同进位位相加，结果存在 A 中	n=0~7 rrr=000~111	1
	ADDC A, direct	00110101 direct	$A \leftarrow A + (direct) + Cy$ 累加器 A 中的内容与直接地址单元的内容连同进位位相加，结果存在 A 中		1
	ADDC A, @Ri	0011011i	$A \leftarrow A + (Ri) + Cy$ 累加器 A 中的内容与工作寄存器 Ri 所指向地址单元中的内容相加，结果存在 A 中（带进位）	i=0, 1	1
	ADDC A, #data	00110100 data	$A \leftarrow A + data + Cy$ 累加器 A 中的内容与工作寄存器 Ri 指向地址单元中的内容、连同进位位相加，结果存在 A 中		1

加法指令中的 ADD 与 ADDC 的区别为是否有进位位 Cy；指令执行结果均在累加器 A 中；以上指令结果均影响程序状态字寄存器 PSW 的 Cy、OV、AC 和 P 标志。

(2) 减法指令

减法指令是将立即数、直接地址、间接地址及工作寄存器与累加器 A 连同借位位 C 内

容相减，结果送回累加器 A 中。影响 Cy、AC、OV、P 标志位，第二操作数允许有寄存器寻址、直接寻址、寄存器间接寻址和立即寻址等寻址方式。

减法加法指令操作码助记符、指令格式以及机器码如表 1-19 所示。

表 1-19 C51 单片机减法指令

指令名称	助记符格式	机器码 (B)	相应操作	指令说明	机器周期
带借位 减法指令	SUBB A, Rn	1001rrr	$A \leftarrow A - Rn - Cy$ 加器 A 中的内容和工作寄存器中的内容、连同借位位相减，结果存在 A 中	$n=0\sim7$ $rrr=000\sim111$	1
	SUBB A, direct	10010101 direct	$A \leftarrow A - (direct) - Cy$ 累加器 A 中的内容与直接地址单元中的内容、连同借位位相减，结果存在 A 中		1
	SUBB A, @Ri	1001011i	$A \leftarrow A - (Ri) - Cy$ 累加器 A 中的内容和工作寄存器 Ri 指向的地址单元中的内容、连同借位位相减，结果存在 A 中	$i=0, 1$	1
	SUBB A, #data	10010100 data	$A \leftarrow A - data - Cy$ 累加器 A 中的内容与立即数、连同借位位相减，结果存在 A 中		1

对借位位 Cy 的状态来说，在进行减法运算中，Cy=1 表示有借位，Cy=0 则无借位。OV=1 声明带符号数相减时，从一个正数减去一个负数结果为负数，或者从一个负数中减去一个正数结果为正数的错误情况。在进行减法运算前，如果不知道借位标志位 C 的状态，则应先对 Cy 进行清零操作。

OV 同样用于判别两个带符号数相减后的差是否超出 8 位带符号数所能表示的范围（-128~+127）。当两个异号数相减时，差的符号与被减数相反，则溢出标志 OV 为 1，结果不正确！例如，被减数为正数，减数为负数，相减后，结果应该是正数，但如果累加器 A 的位 7 为 1，即负数，则表明结果不正确。相减时，如果位 3 位向位 4 位借位，则 AC 为 1；反之为 0。奇偶标志 P 总是体现累加器 A 中“1”的奇偶性，因此 P 也会变化。

由于 C51 指令系统只有带借位的减法指令，因此，当需要执行不带借位的减法指令时，可先通过“CLR C”指令，将进位标志 CY 清 0。

(3) 加 1 指令

加 1 指令也称为增量指令，功能是在原寄存器的内容加 1，结果送回原寄存器。加 1 指令结果通常不影响程序状态字寄存器 PSW。如果原寄存器的内容为 FFH，执行加 1 后，结果就会是 00H。可采用直接、寄存器、寄存器间接寻址等寻址方式。

加 1 指令操作码助记符、指令格式以及机器码如表 1-20 所示。

在 INC direct 这条指令中，如果直接地址是 I/O，其功能是先读入 I/O 锁存器的内容，然后在 CPU 进行加 1 操作，再输出到 I/O 上，这就是“读-修改-写”操作。

(4) 减 1 指令

减 1 指令的作用是把所指的寄存器内容减 1，结果送回原寄存器，若原寄存器的内容为 00H，减 1 后即为 FFH，运算结果不影响任何标志位，只有当操作数是累加器 A 时，才影

响奇偶标志位 P。指令可以采用直接、寄存器、寄存器间接寻址等寻址方式。当直接地址是 I/O 口锁存器时，“读-修改-写”操作与加 1 指令类似。

表 1-20 C51 单片机加 1 指令

指令名称	助记符格式	机器码 (B)	相应操作	指令说明	机器周期
增量指令	INC A	00000100	$A \leftarrow A+1$ 累加器 A 中的内容加 1，结果存在 A 中	影响 PSW 的 P 标志	1
	INC Rn	00001rrr	$Rn \leftarrow Rn+1$ 寄存器 Rn 的内容加 1，结果送回原地 址单元中	$n=0\sim7$ $rrr=000\sim111$	1
	INC direct	0101 direct	$direct \leftarrow direct+1$ 直接地址单元中的内容加 1，结果送 回原地单元中		1
	INC @Ri	0000011i	$(Ri) \leftarrow (Ri)+1$ 寄存器的内容指向的地址单元中的内 容加 1，结果送回原地单元中	$i=0, 1$	1
	INC DPTR	10100011	$DPTR \leftarrow DPTR+1$ 数据指针的内容加 1，结果送回数据 指针中		2

减 1 指令操作码助记符、指令格式以及机器码如表 1-21 所示。

表 1-21 C51 单片机减 1 指令

指令名称	助记符格式	机器码 (B)	相应操作	指令说明	机器周期
减 1 指令	DEC A	00010100	$A \leftarrow A-1$ 累加器 A 中的内容减 1，结果送回累加器 A 中	影响 PSW 的 P 标志	1
	DEC Rn	00011rrr	$Rn \leftarrow Rn-1$ 寄存器 Rn 中的内容减 1，结果送回寄存器 Rn 中	$n=0\sim7$ $rrr=000\sim111$	1
	DEC direct	00010101 direct	$direct \leftarrow direct-1$ 直接地址单元中的内容减 1，结果送回直接地 址单元中		1
	DEC @Ri	0001011i	$(Ri) \leftarrow (Ri)-1$ 寄存器 Ri 指向的地址单元中的内容减 1，结 果送回原地单元中	$i=0, 1$	1

当操作数的初值为 00H 时，减 1 后，结果将变为 FFH。

(5) BCD 码调正指令

用 ADD 指令完成 BCD 码加法运算时，高 4 位或低 4 位大于 1001，以及 AC 或 Cy 有效时，必须加 06H、60H 或 66H 进行校正，才能获得正确的结果。十进制加法调正指令就是为此而设置的。

BCD 码调正指令操作码助记符、指令格式、机器码如表 1-22 所示。

表 1-22 C51 单片机 BCD 加法调正指令

指令名称	助记符格式	机器码 (B)	指令说明	机器周期
BCD 加法调正	DA A	11010100	根据进位标志 Cy、辅助进位标志 AC 以及累加器 A 内容, 将累加器 A 内容转化为 BCD 码形式	1

说明:

- ① 结果影响程序状态字寄存器 PSW 的 Cy、OV、AC 和 P 标志。
- ② BCD (binary coded decimal) 码是用二进制形式表示十进制数, 例如十进制数 45, 其 BCD 码形式为 45H。BCD 码只是一种表示形式, 与其数值没有关系。
- BCD 码用 4 位二进制码表示一位十进制数, 这四位二进制数的权为 8421, 所以 BCD 码又称为 8421 码。十进制数码 0~9 所对应的二进制码如表 1-23 所示。

表 1-23 十进制数码与 BCD 码对应表

十进制数码	0	1	2	3	4	5	6	7	8	9
二进制码	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

- ③ DA A 指令将 A 中的二进制码自动调整为 BCD 码。
 - ④ DA A 指令只能跟在 ADD 或 ADDC 加法指令后, 不适用于减法。
- (6) 乘法指令

C51 提供了 8 位无符号数乘法指令, 功能是把累加器 A 和寄存器 B 中的 8 位无符号整数相乘, 所得到的是 16 位乘积的低位字节放在累加器 A 中, 高位字节放在 B 中。如果积大于 255 (0FFH), 则置位溢出标志 OV; 否则 OV 清 0。进位标志 Cy 总是清 0。

该指令操作码助记符、指令格式、机器码如表 1-24 所示。

表 1-24 C51 单片机乘法指令

指令名称	助记符格式	机器码 (B)	指令说明	机器周期
乘法指令	MUL AB	10100100	$BA \leftarrow A \times B$ 累加器 A 中的内容与寄存器 B 中的内容相乘, 结果高位存 B, 低位存 A	4

C51 没有提供 8 位×16 位、16 位×16 位、16 位×24 位等多字节乘法指令, 只能通过单字节乘法指令完成多字节乘法运算。如果来实现 24 位×16 位乘法可以有如下方法: 为 24 位的被乘数占用 3 字节, 用 CBA 表示; 为 16 位的乘数占用 2 字节, 用 ED 表示, 乘积应该为 40 位。显然 “A×D” 为 16 位, “B×D” 为 24 位, “C×D” 为 32 位; “A×E” 为 24 位, “B×E” 为 32 位, “C×E” 为 40 位, 因此, 可以完成 24 位×16 位运算。

(7) 除法指令

C51 提供了 8 位无符号数除法指令, 功能是把累加器 A 中的 8 位无符号整数除以寄存器 B 中的 8 位无符号整数, 所得商的整数部分存放在累加器 A 中, 而余数存在寄存器 B 中。该指令操作码助记符、指令格式、机器码如表 1-25 所示。

表 1-25 C51 单片机除法指令

指令名称	指令格式	机器码 (B)	指令说明	机器周期
除法指令	DIV AB	10000100	A (商) $\leftarrow A \div B$ B (余数) $\leftarrow A \div B$ 累加器 A 中的内容除以寄存器 B 中的内容, 所得到的商存在累加器 A, 而余数存在寄存器 B 中	4

该指令影响标志位。如果除数 (即寄存器 B) 不为 0, 执行后, 溢出标志 OV、进位标志 Cy 总为 0; 如果除数为 0, 执行后, 结果将不确定, OV 置 1, Cy 仍为 0。AC 保持不变; 奇偶标志 P 位随累加器 A 中 1 的个数变化而变化。

尽管 C51 单片机没有提供 16 位 ÷ 8 位、32 位 ÷ 16 位等多位除法运算指令, 只能借助类似多项式除法运算规则完成多位除法运算。

3. 逻辑运算及移位指令

C51 单片机提供了丰富的逻辑运算指令, 包括逻辑非 (取反)、与、或、异或以及循环移位操作等。逻辑运算和移位指令共有 25 条, 有与、或、异或、求反、左右移位、清 0 等逻辑操作, 有直接、寄存器和寄存器间接寻址方式。这类指令一般不影响程序状态字寄存器 PSW 中的标志位。只有带进位 Cy 循环移位时, 才影响 Cy 和奇偶标志 P。

(1) 循环移位指令

这 4 条指令的作用是将累加器中的内容循环左或右移 1 位, 后两条指令是连同进位位 Cy 一起移位。循环移位指令格式、操作码助记符、机器码等如表 1-26 所示。

表 1-26 C51 单片机循环移位指令指令

指令名称	助记符格式	机器码 (B)	相应操作	机器周期
循环左移	RL A	00100011	累加器 A 中的内容左移一位	1
带进位循环左移	RLC A	00110011	累加器 A 中的内容右移一位, 影响 Cy 标志	1
循环右移	RR A	00000011	累加器 A 中的内容连同进位位 Cy 左移一位	1
带进位循环右移	RRC A	00010011	累加器 A 中的内容连同进位位 Cy 右移一位, 影响 Cy 标志	1

注意: 执行带进位的循环移位指令之前, 必须给 Cy 置位或清 0。

(2) 累加器半字节交换指令

这条指令的功能是将累加器 ACC 的高半字节 (Acc.7~Acc.4) 和低半字节 (Acc.3~Acc.0) 互换。累加器半字节交换指令格式、操作码助记符、机器码等如表 1-27 所示。

表 1-27 C51 单片机累加器半字节交换指令

助记符格式	机器码 (B)	相应操作	指令说明	机器周期
SWAP A	11000100	(A)3~0 \leftrightarrow (A)7~4	累加器中的高、低 4 位互相交换	1

注意: 上面指令结果不影响程序状态字寄存器 PSW 标志。

(3) 逻辑与操作指令

这组指令的功能是在指出的操作数之间执行按位的逻辑与操作, 结果存放在目的操作

数中。如果直接地址是 I/O 地址，则为“读-修改-写”操作。操作数可以采用有寄存器寻址、直接寻址、寄存器间接寻址和立即寻址等寻址方式。逻辑与操作指令格式、操作码助记符、机器码等如表 1-28 所示。

表 1-28 C51 单片机逻辑与操作指令

指令名称	指令格式	机器码 (B)	指令说明	机器周期
逻辑与操作指令	ANL A, direct	01010101	$A \leftarrow A \vee \text{direct}$ 累加器 A 中的内容和直接地址单元中的内容执行与逻辑操作，结果存在寄存器 A 中	1
	ANL A, Rn	01011rrr	$A \leftarrow A \vee Rn$ 累加器 A 的内容和寄存器 Rn 中的内容执行与逻辑操作，结果存在累加器 A 中	1
	ANL A, @Ri	0101011i	$A \leftarrow A \vee (Ri)$ 累加器 A 的内容和工作寄存器 Ri 指向的地址单元中的内容执行与逻辑操作，结果存在累加器 A 中	1
	ANL A, #data	01010100	$A \leftarrow A \vee \#data$ 累加器 A 的内容和立即数执行与逻辑操作，结果存在累加器 A 中	1
	ANL direct, A	01010010	$\text{Direct} \leftarrow \text{direct} \vee A$ 直接地址单元中的内容和累加器 A 的内容执行与逻辑操作，结果存在直接地址单元中	1
	ANL direct, #data	01010011	$\text{direct} \leftarrow \text{direct} \vee \#data$ 直接地址单元中的内容和立即数执行与逻辑操作，结果存在直接地址单元中	2

注意：

- ① 以上指令结果通常影响程序状态字寄存器 PSW 的 P 标志。
- ② 逻辑与指令通常用于将一个字节中的指定位清 0，其他位不变。

(4) 逻辑或操作指令

这组指令的功能是在所指定的操作数之间执行按位的逻辑或操作，结果存到目的操作数中去。操作数可以采用有寄存器寻址、直接寻址、寄存器间接寻址和立即寻址等寻址方式，对输出口 P_i ($i=0, 1, 2, 3$) 和逻辑与指令一样是对端口锁存器内容读出修改。如果直接地址是 I/O 地址，则为“读-修改-写”操作。逻辑或操作指令格式、操作码助记符、机器码等如表 1-29 所示。

表 1-29 C51 单片机逻辑或操作指令

指令名称	助记符格式	机器码 (B)	操作说明	机器周期
逻辑或操作	ORL A, direct	01000101	$A \leftarrow A \vee \text{direct}$ 累加器 A 中的内容和直接地址单元中的内容执行逻辑或操作，结果存在寄存器 A 中	1
	ORL A, Rn	01001rrr	$A \leftarrow A \vee Rn$ 累加器 A 的内容和寄存器 Rn 中的内容执行逻辑或操作，结果存在累加器 A 中	1

指令名称	助记符格式	机器码 (B)	操作说明	机器周期
逻辑或操作	ORL A, @Ri	0100011i	$A \leftarrow A \vee (Ri)$ 累加器 A 的内容和工作寄存器 Ri 指向的地址单元中的内容执行逻辑或操作, 结果存在累加器 A 中	1
	ORL A, #data	01000100	$A \leftarrow A \vee \#data$ 累加器 A 的内容和立即数执行逻辑或操作, 结果存在累加器 A 中	1
	ORL direct, A	01000010	$Direct \leftarrow direct \vee A$ 直接地址单元中的内容和累加器 A 的内容执行逻辑或操作, 结果存在直接地址单元中	1
	ORL direct, #data	01000011	$direct \leftarrow direct \vee \#data$ 直接地址单元中的内容和立即数执行逻辑或操作, 结果存在直接地址单元中	2

注意:

- ① 以上指令结果通常影响程序状态字寄存器 PSW 的 P 标志。
- ② 逻辑或指令通常用于将一个字节中的指定位置 1, 其他位不变。

(5) 逻辑异或操作指令

这组指令的功能是在所指出的操作数之间执行按位的逻辑异或操作, 结果存放到目的操作数中去。操作数可以采用寄存器寻址、直接寻址、寄存器间接寻址和立即寻址等寻址方式。这组指令的作用是将两个单元中的内容执行逻辑异或操作。如果直接地址是 I/O 地址, 则为“读-修改-写”操作。逻辑异或操作指令格式、操作码助记符、机器码等如表 1-30 所示。

表 1-30 C51 单片机逻辑异或操作指令

指令名称	助记符格式	机器码 (B)	相应操作	机器周期
逻辑异或操作	XRL A, direct	01100101	$A \leftarrow A \oplus direct$ 累加器 A 中的内容和直接地址单元中的内容执行逻辑异或操作, 结果存在寄存器 A 中	1
	XRL A, Rn	01101rrr	$A \leftarrow A \oplus Rn$ 累加器 A 的内容和寄存器 Rn 中的内容执行逻辑异或操作, 结果存在累加器 A 中	1
	XRL A, @Ri	0110011i	$A \leftarrow A \oplus (Ri)$ 累加器 A 的内容和工作寄存器 Ri 指向的地址单元中的内容执行逻辑异或操作, 结果存在累加器 A 中	1
	XRL A, #data	01100100	$A \leftarrow A \oplus \#data$ 累加器 A 的内容和立即数执行逻辑异或操作, 结果存在累加器 A 中	1
	XRL direct, A	01100010	$Direct \leftarrow direct \oplus A$ 直接地址单元中的内容和累加器 A 的内容执行逻辑异或操作, 结果存在直接地址单元中	1
	XRL direct, #data	01100011	$direct \leftarrow direct \oplus \#data$ 直接地址单元中的内容和立即数执行逻辑异或操作, 结果存在直接地址单元中	2

注意：

- ① 以上指令结果通常影响程序状态字寄存器 PSW 的 P 标志；
- ② 异或原则是相同为 0，不同为 1。

(6) 清 0 和取反指令

取反指令将累加器中的内容按位取反；清零指令将累加器中的内容清 0。清零和取反指令格式、操作码助记符、机器码等如表 1-31 所示。

表 1-31 C51 单片机清 0 和取反指令

指令名称	指令格式	机器码(B)	操作说明	机器周期
清 0 指令	CLR A	11100100	(A)←00H 累加器中的内容清 0	1
取反指令	CPL A	11110100	(A)←(A) 累加器中的内容按位取反	1

注意：

- ① 清 0 指令中 A 中内容清 0，影响 P 标志；
- ② 取反指令中 A 中内容按位取反，影响 P 标志。

4. 控制转移指令

控制转移类指令的本质是改变程序计数器 PC 的内容，从而改变程序的执行方向，所控制的范围即为程序存储器区间。控制转移指令分为无条件转移指令、条件转移指令和调用/返回指令。

C51 单片机的控制转移指令相对丰富，有可对 64 KB 程序空间地址单元进行访问的长调用、长转移指令，也有可对 2 KB 字节进行访问的绝对调用和绝对转移指令，还有在一页范围内短相对转移及其他无条件转移指令，这些指令的执行一般都不会对标志位有影响。

(1) 无条件转移指令

这组指令执行完后，程序就会无条件转移到指令所指向的地址上去。有可对 64 KB 程序空间地址单元进行访问的长转移指令，也有可对 2 KB 字节进行访问的绝对调用和绝对转移指令。C51 无条件跳转指令操作码助记符、格式、机器码如表 1-32 所示。

表 1-32 无条件跳转指令

指令名称	指令格式	机器码(B)	操作说明	机器周期
长跳转	LJMP addr16	00000010	PC←addr16 程序跳转到地址为 addr16 开始的地方执行	2
绝对无条件跳转	AJMP addr11	a10a9a800001a7~a0	PC10~0←addr11 程序跳转到地址为 PC15~11addr11 开始的地方执行，2 KB 内绝对转移	2
短跳转	SJMP rel	10000000	PC←PC+rel -80H(-128)~7FH(127)短转移	2
间接跳转	JMP @A+DPTR	01110011	PC←A+DPTR 64 KB 内相对转移	2

注意：

- ① 指令结果不影响程序状态字寄存器 PSW；
- ② LJMP 指令可以转移到 64 KB 的程序存储器中的任意位置；
- ③ AJMP 指令转移范围是 2 KB；
- ④ SJMP 指令的转移范围是以本指令的下一条指令为中心的-128~+127 字节以内；
- ⑤ JMP 指令通常用于散转（多分支）程序。

(2) 条件转移指令

C51 单片机提供了满足不同条件的跳转指令来提高编程的效率。程序可利用这组丰富的指令根据当前的条件进行判断，看是否满足某种特定的条件，从而控制程序的转向。C51 单片机条件跳转指令操作码助记符、格式、机器码如表 1-33 所示。

表 1-33 条件跳转指令

指令名称	助记符格式	机器码 (B)	操作说明	机器周期
累加器 A 判 0 指令	JZ rel	0110000	若累加器 A=0，则 PC←PC+rel，否则程序顺序执行	2
	JNZ rel	01110000	若累加器 A≠0，则 PC←PC+rel，否则程序顺序执行	2
比较转移指令	CJNE A, #data, rel	10110100	若累加器 A≠#data，则 PC←PC+rel，否则顺序执行；若累加器 A<#data，则 Cy=1，否则 Cy=0	2
	CJNE Rn, #data, rel	10111rrr	若 Rn≠#data，则 PC←PC+rel，否则顺序执行；若 Rn<#data，则 Cy=1，否则 Cy=0	2
	CJNE @Ri, #data, rel	1011011i	若 (Ri)≠#data，则 PC←PC+rel，否则顺序执行；若 (Ri)<#data，则 Cy=1，否则 Cy=0	2
	CJNE A, direct, rel	10110101	若累加器 A≠(direct)，则 PC←PC+rel，否则顺序执行；若累加器(A)<(direct)，则 Cy=1，否则 Cy=0	2
减 1 非零转移指令	DJNZ Rn, rel	11011rrr	Rn←Rn-1，若 Rn≠0，则 PC←PC+rel，否则顺序执行	2
	DJNZ direct, rel	11010101	(direct) ← (direct)-1，若 (direct) ≠ 0，则 PC ← PC+rel，否则顺序执行	2

注意：

- ① 以上指令结果影响程序状态字寄存器 PSW 的 Cy 标志；
- ② 转移范围与 SJMP 指令相同；
- ③ DJNZ 指令通常用于循环程序中控制循环次数。

(3) 调用和返回指令

在程序设计中，常常出现几个地方都需要进行功能完全相同的处理，只是参数不同而已。为了减少程序编写和调试的工作量，使某一段程序能被公用，于是引进了主程序和子程序的概念，通常把具有一定功能的公用程序作为子程序，在子程序的末尾安排一条返回主程序的指令。在一个程序中，往往在子程序中还会调用别的子程序，称为子程序的嵌套，如图 1-33 所示。

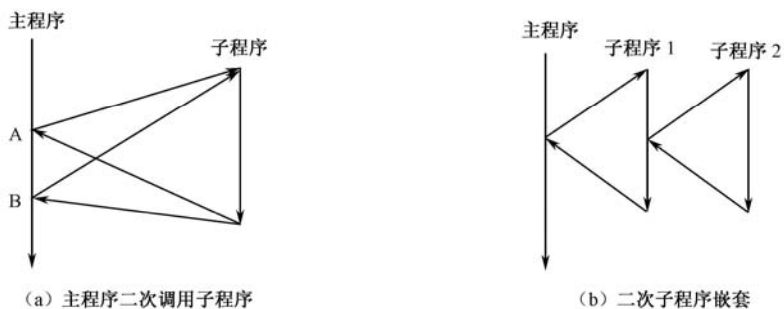


图 1-33 子程序嵌套

C51 单片机调用和返回指令操作码助记符、格式、机器码如表 1-34 所示。

表 1-34 调用和返回指令

指令名称	指令格式	机器码 (B)	相应操作	机器周期
绝对调用指令	ACALL addr11	a10a9a810001 addr7~0	PC←PC+2 SP←SP+1, SP←PC0~7 SP←SP+1, SP←PC8~15 PC0~10←addr11	2
长调用指令	LCALL addr16	00010010 addr15~8 addr7~0	PC←PC+3 SP←SP+1, SP←PC0~7 SP←SP+1, SP←PC8~15 PC←addr16	2
返回指令	RET	00100010	PC8~15←SP, SP←SP-1 PC0~7←SP, SP←SP-1 子程序返回指令	2
中断服务程序 返回指令	RETI	00110010	PC8~15←SP, SP←SP-1 PC0~7←SP, SP←SP-1 中断返回指令	2

指令举例：

ACALL addr11

该指令执行前 PC 值为下一条指令的首地址，转移范围是含有下一条指令首地址的同一个 2KB 范围，即高 5 位地址相同。

LCALL addr16

这条指令无条件地调用位于指定地址的子程序。它先把程序计数器加 3 获得下条指令的地址，并把它压入堆栈（先低位字节后高位字节），并把堆栈指针 SP 加 2。接着把指令的第 2 字节、第 3 字节（a15~a8，a7~a0）分别装入 PC 的高位字节和低位字节中，将从该地址开始执行程序。

LCALL 指令可以调用 64 KB 范围内程序存储器中的任何一个子程序，执行后不影响任何标志。

RET

返回指令是使 CPU 从子程序或中断服务程序返回到主程序执行。指令的功能是从堆栈中退出 PC 的高位字节和低位字节，把栈指针 SP 减 2，并从产生的 PC 值开始执行程序，不影响任何标志。

RETI

中断返回指令，除了执行 RET 指令的功能外，还清除内部相应的中断状态寄存器（该触发器由 CPU 响应中断时置位，指示 CPU 当前是否在处理高级或低级中断）。因此，中断服务程序必须以 RETI 为结束指令。CPU 执行 RETI 指令后至少再执行一条指令，才能响应新的中断请求。

第 2 章 Keil 8051 C 编译器

单片机的开发离不开必要的硬件，同样也离不开软件，编写的汇编语言源程序要变为 CPU 可以执行的机器码有两种方法，一种是手工汇编，另一种是机器汇编。目前已极少使用手工汇编的方法了，机器汇编是通过汇编软件将源程序变为机器码，用于 C51 单片机的汇编软件有早期的 A51，随着单片机开发技术的不断发展，从普遍使用汇编语言到逐渐使用高级语言开发，单片机的开发软件也在不断地发展，Keil 软件是目前最流行开发 C51 单片机的软件。

Keil C51 标准 C 编译器为 8051 微控制器的软件开发提供了 C 语言环境，同时保留了汇编代码高效、快速的特点。C51 编译器的功能不断增强，可以更加贴近 CPU 本身及其他的衍生产品。

C51 已被完全集成到 μ Vision3 的集成开发环境中， μ Vision3 IDE 可为它们提供单一而灵活的开发环境。

2.1 系统概述

Keil C51（简称为 C51）是美国 Keil Software 公司研制的 51 系列兼容单片机 C 语言软件开发系统，与汇编相比，C 语言在功能、结构性、可读性、可维护性等方面有明显的优势，而且易学易用。用过汇编语言后再使用 C 语言来开发，体会将更加深刻。

Keil C51 软件提供丰富的库函数和功能强大的集成开发调试工具，全 Windows 界面。另外，重要的一点是，只要看一下编译后生成的汇编代码，就能体会到 Keil C51 生成的目标代码的效率是非常高的，多数语句生成的汇编代码非常紧凑，容易理解。在开发大型软件时更能体现高级语言的优势。

Keil C51 提供了包括 C 编译器、宏汇编、连接器、库管理和一个功能强大的仿真调试器等在内的完整开发方案，通过一个集成开发环境（ μ Vision）将这些部分组合在一起。

运行 Keil C51 软件需要 Pentium 或以上的 CPU、16 MB 或更多 RAM、20 MB 以上的硬盘空间、WIN98、NT、WIN2000、WINXP 等操作系统。

下面详细介绍 Keil C51 开发系统各部分的功能和使用。

当使用 Keil C51 工具时，项目开发流程和其他软件开发项目的流程极其相似，基本流程如下：

- ① 创建一个项目从器件库中选择目标器件配置工具设置；
- ② 用 C 语言或汇编语言创建源程序；
- ③ 用项目管理器生成应用程序；
- ④ 修改源程序中的错误；
- ⑤ 测试链接应用。

一个完整的 8051 工具包的框图可以表述此开发流程每一个组件，C51 工具包的整体结

构，如图 2-1 所示。

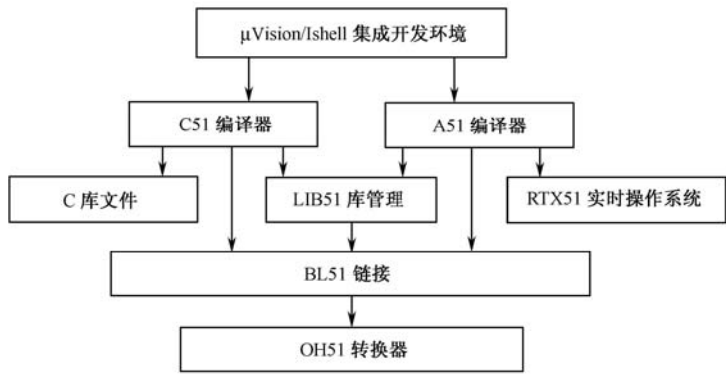


图 2-1 C51 工具包的整体结构图

其中 μVision 与 Ishell 分别是 C51 针对 Windows 和针对 DOS 的集成开发环境（IDE），可以完成编辑、编译、链接、调试、仿真等整个开发流程。开发人员可用 IDE 本身或其他编辑器编辑 C 文件或汇编源文件，然后分别由 C51 及 A51 编译器编译生成目标文件（OBJ）。目标文件可由 LIB51 创建生成库文件，也可以与库文件一起经 L51 链接定位生成绝对目标文件（ABS）。ABS 文件由 OH51 转换成标准的 HEX 文件，以供调试器 dScope51 或 tScope51 使用进行源代码级调试，也可由仿真器使用直接对目标板进行调试，也可以直接写入程序存储器，如 EPROM 中。

1. μVision3 IDE

μVision3 集成开发环境集成了一个项目管理器、一个功能丰富有错误提示的编辑器，以及设置选项生成工具，可以很容易地利用 μVision3 创建源代码并把它们组织到一个能确定目标应用的项目中去。

2. C51 编译器和 A51 汇编器

源代码由 μVision3 IDE 创建，并被 C51 编译或 A51 汇编编译器和汇编器从源代码生成可重定位的目标文件，C51 编译器完全遵照 ANSI C 语言标准，支持 C 语言的所有标准特性，另外直接支持 8051 单片机结构的几个特性被添加到里面。Keil A51 宏汇编器支持 8051 单片机及其衍生系列的全部指令集。

3. LIB51 库管理器

LIB51 库管理器允许从由编译器或汇编器生成的目标文件创建目标库，是一种被特别组织过并在以后可以被链接重用的对象模块，当链接器处理一个库时，仅仅那些被使用的目标模块才被真正使用。

4. BL51 链接器/定位器

BL51 链接器/定位器利用从库中提取的目标模块和由编译器或汇编器生成的目标模块，创建一个绝对地址的目标模块。一个绝对地址目标模块或文件包含不可重定位的代码和数据

所有的代码和数据，被安置在固定的存储器单元中。

该绝对地址目标文件可以用来：

- 写入 EPROM 或其他存储器件；
- 由 μ Vision2 调试器使用来模拟和调试；
- 由仿真器用来测试程序。

5. μ Vision3 调试器

μ Vision3 源代码级调试器是一个理想的、快速的、可靠的程序调试器，它包含 1 个模拟器，能够模拟整个 8051 系统，包括片上外围器件和外部硬件。当从器件选择器件时，这个器件的特性将自动配置。

μ Vision3 调试器在实际目标板上测试程序时提供了以下两种方法：

- ① 安装 MON51 目标监控器到目标系统并且通过 Monitor-51 接口下载程序；
- ② 利用高级的 GDIAGDI 接口，把 μ Vision3 调试器绑定到目标系统中。

6. RTX51 实时操作系统

RTX51 实时操作系统是一个针对 8051 系列的多任务核。RTX51 实时内核从本质上简化了对实时事件反应速度要求高的复杂应用系统的设计、编程和调试。RTX51 实时内核完全集成到 C51 编译器中，从而方便使用。任务描述表和操作系统的链接由 BL51 连接器/定位器自动控制。

2.2 使用Keil开发

Keil μ Vision3 是一套在 Windows 环境下，8051 单芯片整合性开发接口（integrated development environment, IDE）软件，它具备完善的项目管理系统，提供编辑器以编写程序及说明文件，可以协助编写、翻译（包含 C 语言的编译器 C51 Compiler 以及 A51 组译者）、除错和测试嵌入式系统程序（embedded programs）。

2.2.1 μ Vision3 项目管理窗口简介

1. μ Vision3 的项目管理与程序编辑

启动 Keil μ Vision3 应用程序，窗口画面就会显示 μ Vision3 窗口。由于 μ Vision3 是以项目（project）管理整合作业，如果安装完后第一次使用时，若无项目加载，许多功能画面无法展示。

因此为了要观察 μ Vision3 整合环境画面，主要操作步骤如下：

① 执行 Project/Open Project 菜单命令，弹出“Select Project File”对话框，如图 2-2 所示。

② 选取 C:\Keil\C51\Examples\ASM\ASAMPLE 的范例项目，开启此项目，如图 2-3 所示。

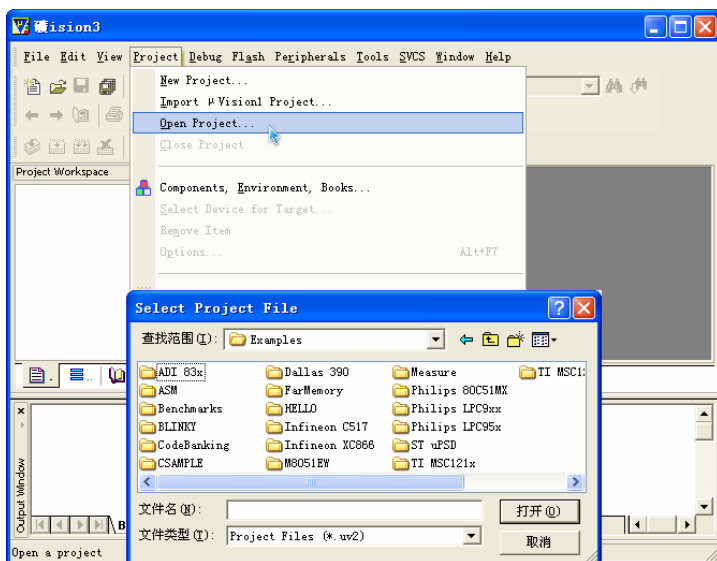


图 2-2 打开项目文件

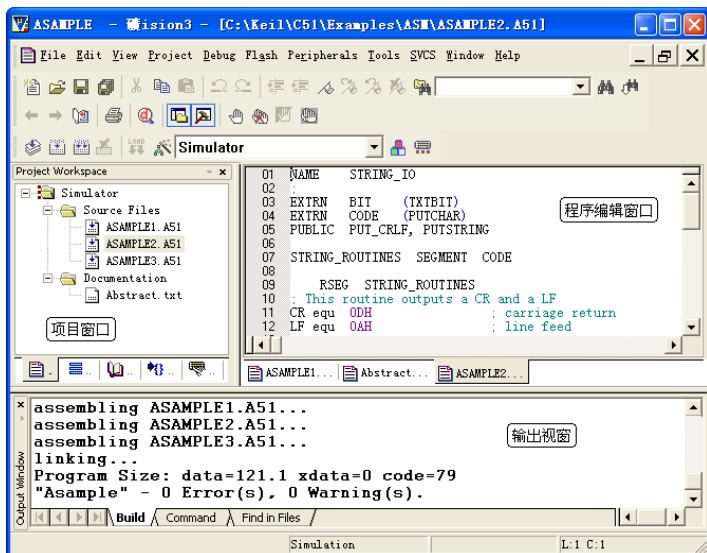


图 2-3 加载项目后的窗口画面

③ 利用 View 菜单内的菜单指令，观察对应的子窗口 Project Winodws、Output Windows，工具列 Status Bar、File Toolbar、Build Toolbar、Build Toolbar，如图 2-4 所示。

View 内的菜单命令的描述如下所示：

- Status Bar: 显示/隐藏状态条；
- File Toolbar: 显示/隐藏文件菜单条；
- Build Toolbar: 显示/隐藏编译菜单条；
- Debug Toolbar: 显示/隐藏调试菜单条；
- Project Window: 显示/隐藏项目窗口；

- Output Window: 显示/隐藏输出窗口;
- Source Browser: 打开资源浏览器。

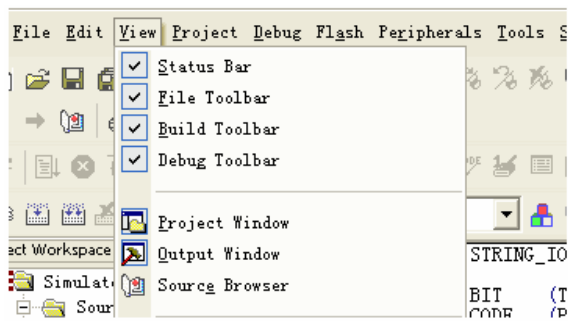


图 2-4 View 菜单内容对应的工具列

经过上述的操作，可以看到几个在 Keil μVision3 整合环境中重要的部分，如负责项目管理的项目窗口、可编辑 8051 汇编语言与 C 语言程序文件的程序编辑窗口、显示执行结果的输出窗口，与经常用到的工具列。

2. 项目窗口

进入 Keil μVision3 后，在窗口画面左侧可发现如图 2-5 所示的项目窗口。

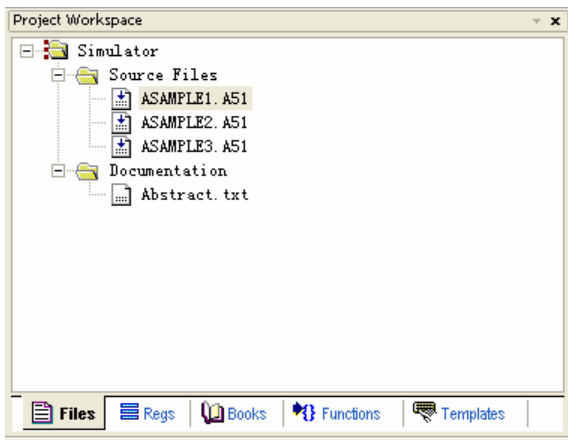


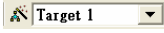
图 2-5 项目窗口

若项目窗口没有出现，则执行 View/Project Window 菜单命令，将它开启，或者执行 View/New Project 菜单命令，新增一个项目。

项目窗口又可分为 File、Regs、Books、Functions、Templates 共 5 种页面，可单击窗口下方的标示进行页面的切换，其中，Regs 是在进入调试功能时会显示 8051 的缓存器状态；Books 显示 μVision3 的在线操作说明书；Files 负责管理项目的所有档案。

通常在开发较大的程序时，会将不同作用的功能各别写在不同程序中，项目窗口的 File 页面就是用来管理这些档案程序。又可分为三个层级：Target→Group→File，说明如下。

① **Target:** 在同一个项目中，可以拥有一个以上的 **Target**，两个 **Target** 之间基本上可以共享相同的原始程序档案，但是可以各自有着不同的输出设定或不同的装置，通过建立不同的 **Target**，可以得到不同的输出程序版本。

如果有两个以上的 **Target**，可利用工具列  切换与设定不同的 **Target**。若要新增 **Target**，在项目窗口的{**Target**}单击鼠标右键，选取[**Target Group File**]指令。

② **Group:** 通常希望将众多的档案分门别类地安置好，以方便自己或其他人方便查阅管理，因此可以在 **Target** 下建立多个 **Group**，将有相同性质的原始程序代码或文件归类在同一个 **Group**，例如，可以建立一个 **I/O Group**，将所有有关输入输出的程序文件放于其中。

③ **File:** 在每个 **Group** 之下，可以加入不同的档案。例如，**C51** 原始程序文件 (*.c)、**A51** 汇编语言程序 (*.a 或 *.src)，已编译的对象程序文件 (*.obj)，链接库档案 (*.lib) 以及纯文本文件 (*.txt)。

若要在项目中加入文件，在 **Source Files** 图标上单击鼠标右键，在快速选单上单击选中 **Add File to Group ‘Source Files’**，选择要加入的文件；若要移除档案，在文件名称上单击鼠标右键，单击选中 **Remove File ‘文件名’**，移除该文件；要在 **μVision2** 中开启项目中的档案，只要双击选项目管理子窗口上的档案即可。

3. 在线操作说明书

在线说明文件 (Get Help): **μVision3** 提供了完整的操作说明，如果想查阅其内容，则可以在“项目管理”窗口下方单击 **Book** 选项，再双击想阅读的子标题即可，如图 2-6 所示。

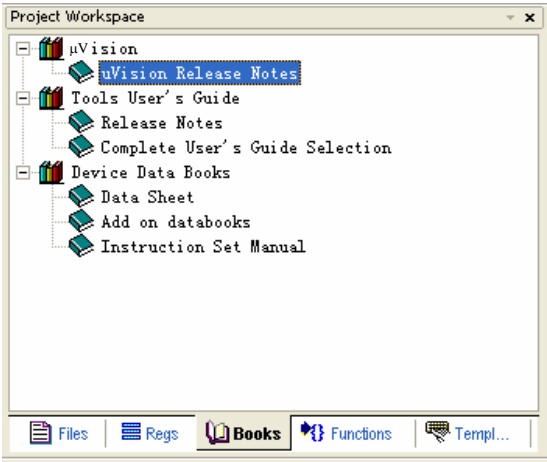


图 2-6 项目管理窗口的 Books 页面

此外，在操作过程中若有疑问处，将光标留在该处，按下 **F1** 键，出现辅助说明窗口，即可取得更多的信息。如图 2-7 所示的在程序编译窗口中的 **END** 处按下 **F1** 键后弹出的辅助说明窗口。

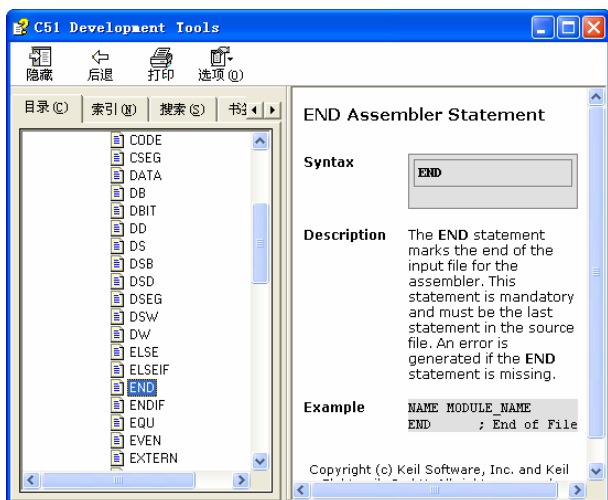


图 2-7 在线辅助说明

2.2.2 Keil C51 开发过程

以下是 Keil μ Vision3 开发过程。

① 新建或开启项目。新建的项目必须设定 Target 的 Device，即选用芯片的公司与型号。

② 在项目中加入程序文件。项目内的程序文件来源有两种方式：

- 将已经存在的程序文件直接加入项目；
- 选择 File 菜单的 New 指令，建立一个新档案，再将此档案加入项目。

存盘时要注意的，若准备采用 C51 编译器，档案的附属名称使用 “*.c”，若使用 A51 编译器，档案的附属名称使用 “*.a”。

③ 编辑/修改程序。已加入项目的程序文件，可以直接在项目窗口中双击该程序文件的文件名，即可开启该档案的程序编辑窗口。若未加入项目的程序文件，执行 File/Open 菜单指令，加载程序编辑窗口，进行编辑与修改。

④ 编译与链接。 μ Vision3 同时提供了 C51 这个兼容于 ANSI C 标准的编译器 (C51 compiler)，以及 A51 这个编译器 (A51 assembler)，可以用这两种程序语言，开发单芯片程序。当原始程序都被编译完成后，再通过链接以建立最后可执行的程序，在 μ Vision3 附有 BL51 这个链接器来协助完成这项工作。

在链接的过程中，标准 C 语言的函数库 (ANSI C standard library) 也会被汇入。另外，使用 CPU 多任务处理的开发者也会使用到 RTX51 (real-time operator system) 这个实时操作系统的链接库，也可以通过 LIB51 这个链接库管理自己编写的链接库。

⑤ 调试与在线仿真。编译与链接的机器码程序，除了可以直接烧录到芯片上，也可以通过 μ Vision3 提供的储存工具彻底地消除程序中的错误并且最佳化程序，还可以借助平行仿真系统在 PC 上直接仿真硬件上的各种操作，使程序更加可靠。

下面将详细进行描述。

1. 启动μVision3 并创建一个项目

μVision3 是一个标准 Windows 应用程序，直接单击程序图标就可以启动它。要想新建一个项目文件，可以执行 Project/New Project 菜单指令，在建立项目对话框中输入“keil”项目名，进入 Device Database，在 CPU 选项中点选“Atmel”的 AT89C51，如图 2-8 所示。

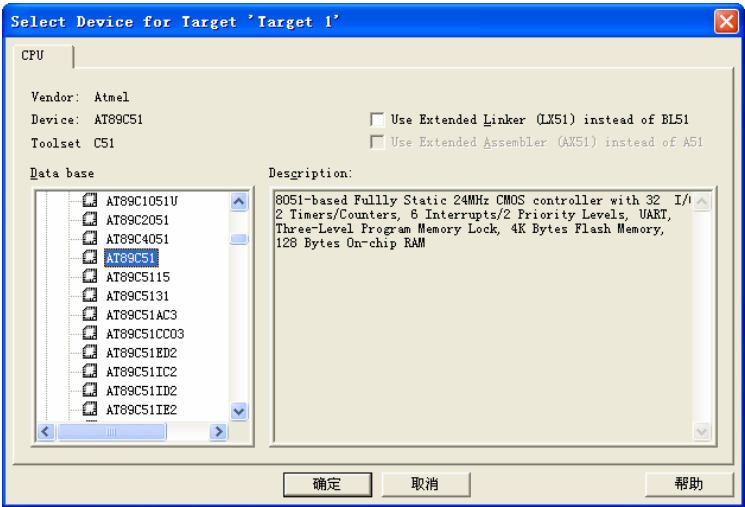


图 2-8 设定窗口

单击 **确定** 按钮，进入μVision3 环境中，在项目管理窗口的 Target1，单击前面的“+”号，打开 Source Group1 目录，里面还没有任何程序文件。

2. 新建一个源文件

可以执行 File/New 菜单命令新建一个源文件，打开一个空的编辑窗口，用来输入源代码。当把此文件另存为“*.c”文件后，编辑器将高亮显示 C 语言语法字符。

创建了源文件后，需要将它加入到建立的项目中，在 Source Group 1 图标上单击鼠标右键，在快速选单选择 Add File to Group ‘Source Group 1’，选择要加入的文件 main.c。

3. 工程设置

工程建立好以后，还要对工程进行进一步的设置，以满足要求。

① 首先单击左边项目管理窗口的 Target 1，然后执行 Project/Options for target ‘Target 1’ 菜单命令，即可出现对工程设置的对话框，如图 2-9 所示。

② 弹出“Options for Target ‘Target 1’ ”对话框，切换到 Target 选项卡，可对项目目标选项进行设置，主要是 Xtal（MHz）、Operating；ROM 和 RAM 的设置，如图 2-10 所示。

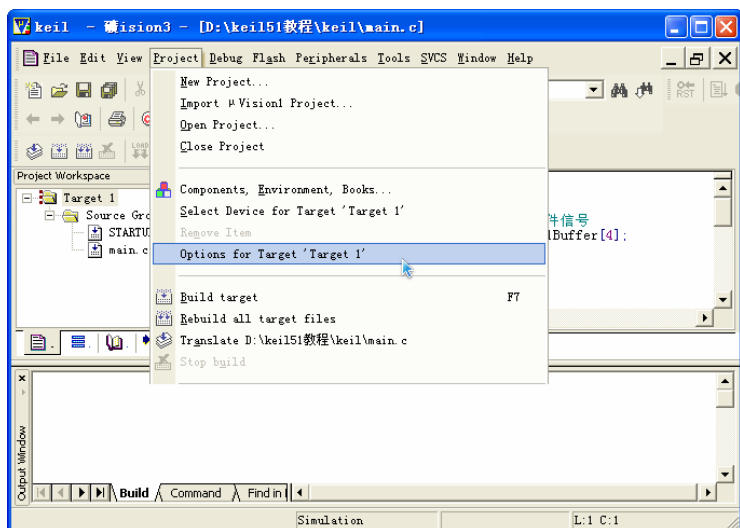


图 2-9 执行菜单命令

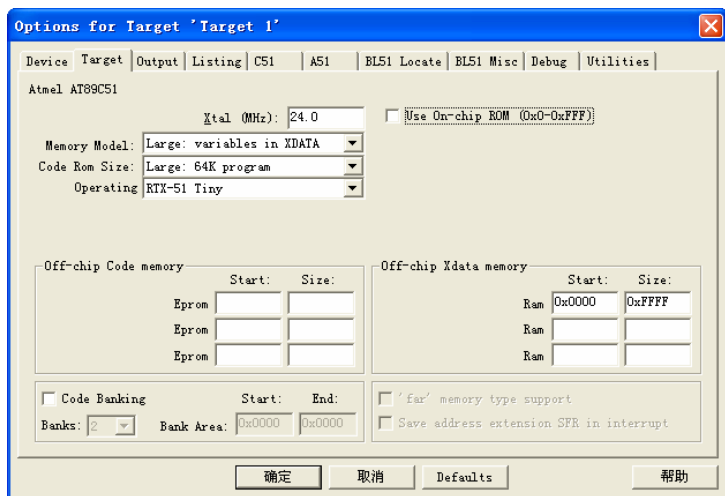


图 2-10 项目目标选项的设置

下面描述了目标对话框的一些选项：

- Xtal 后面的数值是晶振频率值，默认值是所选目标 CPU 的最高可用频率值，对于 AT89C51 而言是 24.0 MHz，该数值与最终产生的目标代码无关，仅用于软件模拟调试时显示程序执行时间。正确设置该数值可使显示时间与实际所用时间一致，一般将其设置成与硬件所用晶振频率相同，但没必要了解程序执行的时间。
- Memory Model 用于设置 RAM 使用情况，有三个选择项，Small 表示所有变量都在单片机的内部 RAM 中；Compact 表示可以使用一页外部扩展 RAM；Larget 表示可以使用全部外部的扩展 RAM。
- Code Rom Size 用于设置 ROM 空间的使用，同样也有三个选择项，Small 模式，只用于低于 2 KB 的程序空间；Compact 模式，单个函数的代码量不能超过 2 KB，整个程序

可以使用 64 KB 程序空间；Larget 模式，可用全部 64 KB 空间。

- Operating 项是操作系统选择，Keil 提供了两种操作系统：RTX51 Tiny 和 RTX51 Full。
- Off-chip Code memory 用以确定系统扩展 ROM 的地址范围；Off-chip Xdata memory 组用于确定系统扩展 RAM 的地址范围，这些选择项必须根据所用硬件来决定，由于该例是单片应用，未进行任何扩展，所以均不重新选择，按默认值设置。

③ 对这个工程进行设置，让它来输出所需要的 HEX 文件，因为 Keil 默认的输出是没有 HEX 文件的。切换到 Output 选项卡，可进行一些编译输出信息的设置，在里面的 Create HEX File 前，选中打上勾，这样可以 Let Keil 输入烧录所需的 HEX 文件，如图 2-11 所示。

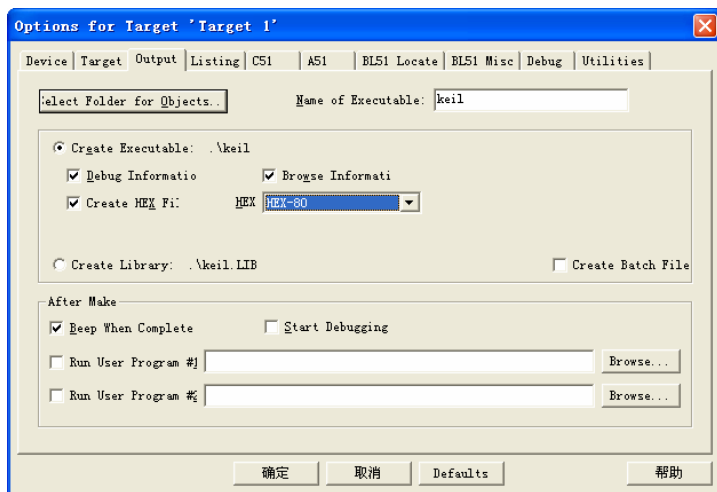


图 2-11 项目输出选项的设置

这里面也有多个选择项，其中 Create HEX File 用于生成可执行代码文件（可以用编程器写入单片机芯片的 HEX 格式文件，文件的扩展名为 HEX），默认情况下该项未被选中，如果要写片做硬件实验，就必须选中该项。

选中 Debug Information 将会产生调试信息，这些信息用于调试，如果需要对程序进行调试，应当选中该项。

Browse Information 是产生浏览信息，该信息可以执行 View/Browse 菜单命令来查看，这里设置为默认值。

按钮 Select Folder for Objects 是用来选择最终的目标文件所在的文件夹，默认值是与工程文件在同一个文件夹中。

Name of Executable 用于指定最终生成的目标文件的名字，默认与工程的名字相同，这两项一般不需要修改。

工程设置对话框中的其他各页面与 C51 编译选项、A51 的汇编选项、BL51 链接器的链接选项等用法有关，这里均取默认值，不进行任何修改。

下面仅对一些有关页面中常用的选项进行简单介绍。

Listing 标签页用于调整生成的列表文件选项。在汇编或编译完成后将产生 (*.lst) 的列表文件，在链接完成后也将产生 (*.m51) 的列表文件，该页用于对列表文件的内容和形式

进行细致地调节，其中比较常用的选项是 C Compile Listing 下的 Assamble Code 项，选中该项可以在列表文件中生成 C 语言源程序所对应的汇编代码。如图 2-12 所示。

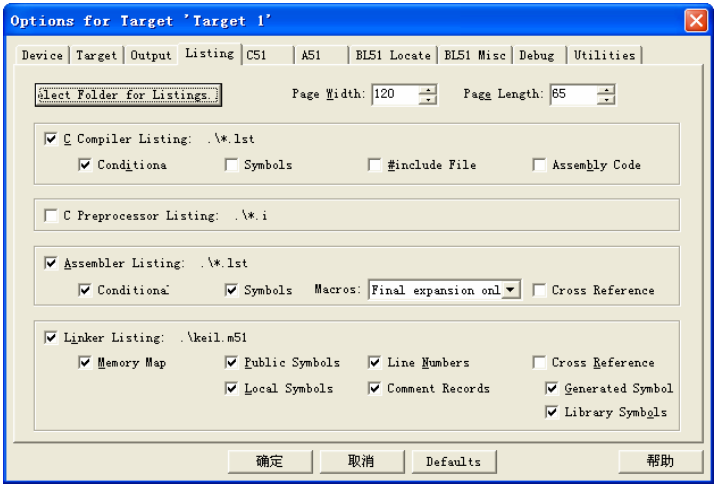


图 2-12 Listing 标签页设置

C51 标签页用于对 Keil 的 C51 编译器的编译过程进行控制，其中比较常用的是 Code Optimization 组，如图 2-13 所示。该组中 Level 是优化等级，C51 在对源程序进行编译时，可以对代码多至 9 级优化，默认为第 8 级，一般不必修改，如果在编译中出现一些问题，可以降低优化级别。Emphasis 是选择编译优先方式，第一项是代码量优化（最终生成的代码量小）；第二项是速度优先（最终生成的代码速度快）；第三项是默认。默认的是速度优先，可根据需要更改。

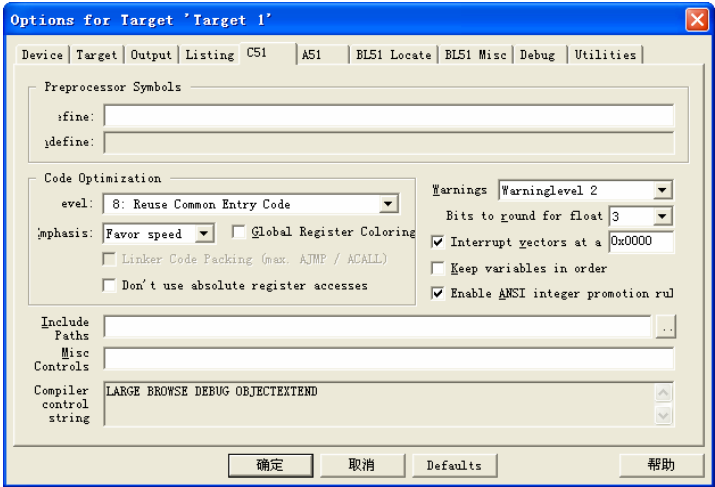


图 2-13 “C51” 标签页设置

④ 设置完成后按确认返回主界面，工程文件建立、设置完毕。

4. 编译、链接

在设置好工程后，即可进行编译、链接。

执行 Project/Build target 菜单命令，可对当前工程进行链接。如果当前文件已修改，软件会先对该文件进行编译，然后再链接以产生目标代码；如果执行 Project/Rebuild all target files 菜单命令，将会对当前工程中的所有文件重新进行编译和链接，确保最终生产的目标代码是最新的；而 Translate...选项仅对该文件进行编译，而不进行链接。如图 2-14 所示的 Project 菜单选项。

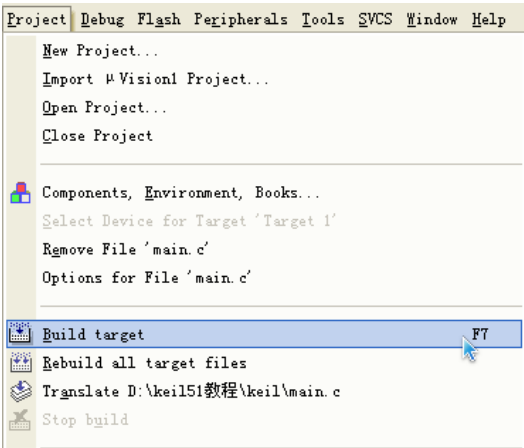


图 2-14 Project 菜单选项

以上操作也可以通过工具栏按钮直接进行。
图 2-15 是有关编译、设置的工具栏按钮，从左到右分别是编译、编译链接、全部重建、停止编译和对工程进行设置。



图 2-15 Build 工具栏

编译过程中的信息将出现在 Output Window（输出窗口）中的 Build 页中，如果使用者编写程序有错误，可以从窗口输出得知。错误信息会显示程序行号，让用户知道错误在哪里。也可以直接在错误信息那一行双击，程序编辑器会在该行前面显示蓝色箭头。程序编译要确认无误才能进行调试，即 Output Window（输出窗口）出现如图 2-16 所示的提示信息时。一般有警告信息（Warning）时，仍然可以进行调试。

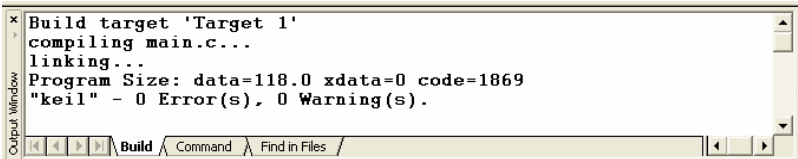


图 2-16 程序编译提示信息

程序编译无误后，即可获得可执行的*.HEX 文件，该文件可以被编程器读入并写到芯片中，同时还产生了一些其他相关的文件，可被用于 Keil 的仿真与调试，这时可以进入下一步调试的工作。

2.2.3 Keil的调试

学习了如何建立工程、汇编、链接工程，并获得目标代码，但是做到这一步仅仅代表源程序没有语法错误，至于源程序中是否存在着的其他错误，必须通过调试才能发现并解决。

事实上，除了简单的程序外，绝大部分的程序都要经过反复调试才能得到正确的结果，因此，调试是软件开发中重要的一个环节，本节将介绍常用的调试命令、利用在线汇编、各种设置断点进行程序调试的方法。

1. 启动调试

当源程序代码编译成功后，运行 dScope，可以对 8051 应用程序进行软件仿真调试。运行设置对话框中选择左侧的 Use Simulator 选项，具体设置如图 2-17 所示。

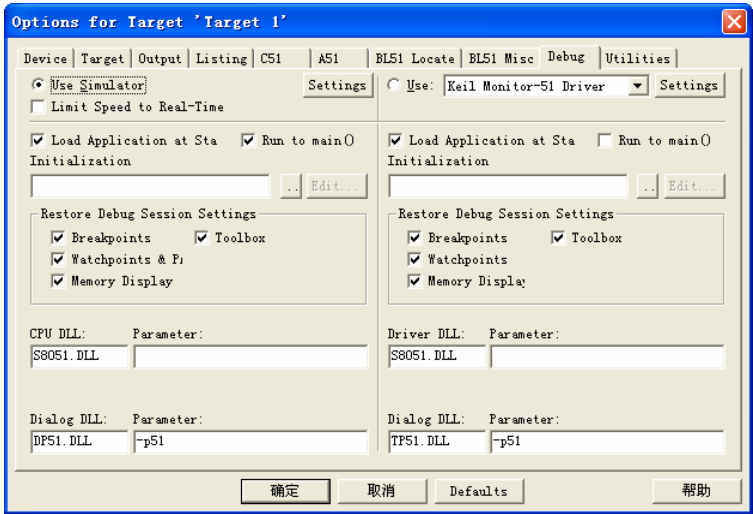


图 2-17 Debug 标签页面设置

Load Application at Startup 选项用于在 dScope 开始时调用自己应用程序的 OMF 文件，因此要选中这个复选框。

Run to main()选项用于选择在 dScope 开始后从 C 语言源程序的 main()函数开始执行，因此选中此复选框。

在 Keil 执行菜单中，单击带有红色 d 字母的按钮，就可以启动 dScope 了，如图 2-18 所示。

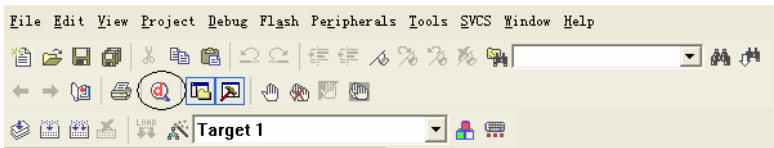



图 2-18 Keil 执行菜单

2. 常用调试命令

执行 **Debug Start/Stop Debug Session** 菜单命令，或者单击快捷键图标 ，即可进入调试状态，Keil 内部建了一个仿真 CPU，用来模拟执行程序，该仿真 CPU 功能强大，可以在没有硬件和仿真机的情况下进行程序的调试，下面将要学的就是该模拟调试功能。不过在学习之前必须明确模拟毕竟只是模拟，与真实的硬件执行程序肯定还是有区别的，其中最明显的就是时序，软件模拟不可能和真实的硬件具有相同的时序，具体的表现就是程序运行的速度和使用的计算机有关，计算机性能越好，运行速度越快。

进入调试状态后，界面与编辑状态相比有了明显的变化。**Debug** 菜单项中原来不能用的命令现在已可以使用了，工具栏会多出一个用于运行和调试的工具条，如图 2-19 所示。



图 2-19 调试工具条



Debug 菜单上大部分的命令可以在此工具条找到对应的快捷按钮，从左到右依次是复位、运行、暂停、单步、过程单步、执行完当前子程序、运行到当前行、下一状态、打开跟踪、观察跟踪、反汇编窗口、观察窗口、代码作用范围分析、1# 串行窗口、内存窗口、性能分析命令。

学习程序调试，必须明确两个重要的概念，即单步执行与全速运行。

全速执行是指一行程序执行完以后紧接着执行下一行程序，中间不停止，这样程序运行的速度很快，并可以看到该段程序执行的总体效果，即最终结果正确还是错误。如果程序有错，则难以确认错误出现在程序的哪些行。

单步执行是每次执行一行程序，执行完该行程序后即停止，等待执行下一行程序的命令，此时可以观察该行程序执行完后的结果，可以判断与该行程序所想要得到的结果是否相同，借此可以找到程序中问题所在。

程序调试中，这两种运行方式都要用到。

执行 **Debug/Step** 菜单命令，或相应的快捷按钮 ，或使用快捷键 **F11**，可以单步执行程序。执行 **Debug/Step Over** 菜单命令，或相应快捷按钮 ，或使用快捷键 **F10**，可以以过程单步形式执行命令，所谓过程单步，是指将汇编语言中的子程序或高级语言中的函数作为一个语句来全速执行。

按下 **F11** 键，可以看到源程序窗口的左边出现了一个黄色调试箭头，指向源程序的第一行，如图 2-20 所示。

每按一次 **F11** 键，执行该箭头所指的程序行，然后箭头指向下一行程序，当箭头指向 **USB_Delay1ms** 行时，再次按下 **F11** 键，会发现进行 **for** 语句的循环操作。通过单步执行程序，可以找出一些问题的所在，但是仅依靠单步执行来查错有时是困难的，或虽能查出错误但效率很低，为此必须辅之以其他的方法。

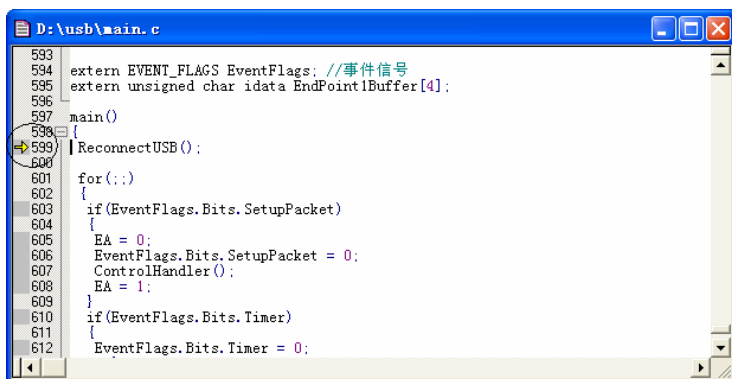


图 2-20 调试窗口

3. 在线汇编

在进入 Keil 的调试环境以后，如果发现程序有错，可以直接对源程序进行修改，但是要修改后的代码起作用，必须先退出调试环境，重新进行编译、链接后再次进入调试，如果只是需要对某些程序行进行测试，或仅需对源程序进行临时的修改，这样的过程未免有些麻烦，为此 Keil 软件提供了在线汇编的能力，将光标定位于需要修改的程序行上，执行 Debug/Inline Assembly... 菜单命令，即可弹出如图 2-21 所示的对话框。

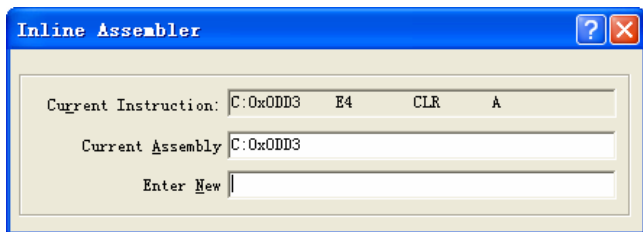



图 2-21 在线汇编对话框




在 Enter New 后面的编辑框内直接输入需更改的程序语句，输入完后按回车键将自动指向下一条语句，可以继续修改，如果不再需要修改，可以单击右上角的关闭按钮关闭窗口。

4. 断点设置

程序调试时，一些程序行必须满足一定的条件才能被执行到（例如，程序中某变量达到一定的值、按键被按下、串口接收到数据、有中断产生等），这些条件往往是异步发生或难以预先设定的，这类问题使用单步执行的方法是很难调试的，这时就要使用到程序调试中的另一种非常重要的方法——断点设置。

断点设置的方法有多种，常用的是在某一程序行设置断点，设置好断点后可以全速运行程序，一旦执行到该程序行即停止，可在此观察有关变量值，用以确定问题所在。

在程序行设置/移除断点的方法是将光标定位于需要设置断点的程序行，执行菜单 Debug/Insert/Remove BreakPoint 命令，设置或移除断点（也可以用鼠标在该行双击实现同样的功能），快捷按钮为 ；执行菜单 Debug/Enable/Disable Breakpoint 命令，可以开启或暂

停光标所在行的断点功能，快捷按钮为；执行菜单 Debug/Disable All Breakpoint 命令，可以暂停所有断点，快捷按钮为；执行菜单 Debug/Kill All BreakPoint 命令，可以清除所有的断点设置，快捷按钮为。

除了在某程序行设置断点这一基本方法以外，Keil 软件还提供了多种设置断点的方法，执行 Debug/Breakpoints...菜单命令，即可出现一个对话框，该对话框用于对断点进行详细地设置，如图 2-22 所示。

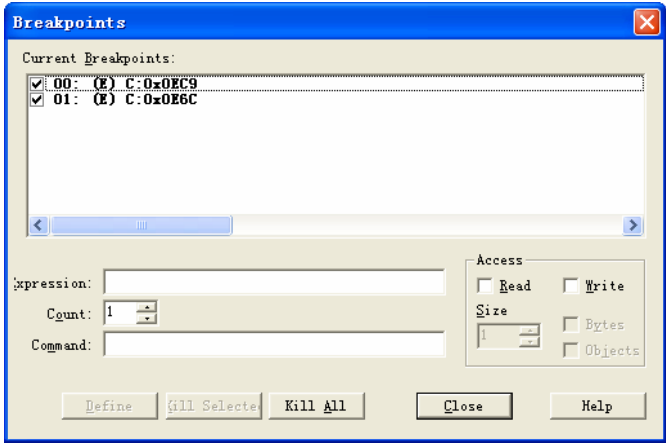


图 2-22 断点设置对话框

图 2-22 中 Expression 后的编辑框内用于输入表达式，该表达式用于确定程序停止运行的条件，这里表达式的定义功能非常强大，涉及到 Keil 内置的一套调试语法，这里不进行详细说明。

下面举例进行说明。

① 在 Expression 中键入“a==0xf7”，再单击 Define 按钮，即定义了一个断点，注意，a 后有两个等号，意为相等。该表达式的含义是：如果 a 的值到达 0xf7，则停止程序运行。除使用相等符号之外，还可以使用>、>=、<、<=、!=（不等于）、&（两值按位与）、&&（两值相与）等运算符号。

② 在 Expression 中键入“Delay”，再单击 Define 按钮，其含义是如果执行标号为“Delay”的行则中断。

③ 在 Expression 中键入“Delay”，按 Count 后的微调按钮，将值设置为 3，其意义是当第 3 次执行到“Delay”时才停止程序运行。

④ 在 Expression 中键入“Delay”，在“Command”后键入“printf（“SubRoutine ‘Delay’ has been Called\n”）”，表示主程序每次调用“Delay”程序时并不停止运行，但会在输出窗口 Command 页输出一行字符，即“SubRoutine ‘Delay’ has been Called”，其中“\n”表示回车换行，使窗口输出的字符换行对齐。

⑤ 设置断点前先在输出窗口的“Command”页中键入 DEFINE int I，然后在断点设置时同上一条，但是“Command”后键入“printf（“SubRoutine ‘Delay’ has been Called %dtimes\n”，++I）”，则主程序每次调用“Delay”时将会在“Command”窗口输出该字符及被调用的次数，如“SubRoutine ‘Delay’ has been Called 10 times”。

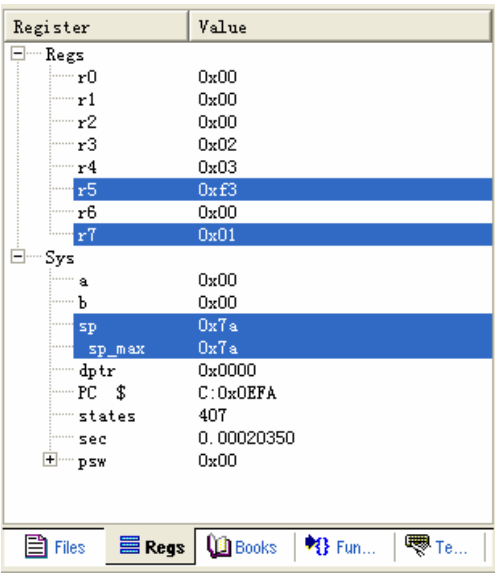
对于使用 C 语言源程序的调试，表达式中可以直接使用变量名，但必须要注意，设置时只能使用全局变量名和调试箭头所指模块中的局部变量名。

5. 调试窗口

Keil 软件在调试程序时提供了多个窗口，主要包括工程窗口（Regs Window）、输出窗口（Output Window）、观察窗口（Watch & Call Stack Window）、存储器窗口（Memory Window）、反汇编窗口（Disassembly Window）、串行窗口（Serial Window）等。进入调试模式后，通过菜单 View 下的相应命令可以打开或关闭这些窗口。

（1）工程窗口（Regs Window）

项目管理窗口下方，选择 Regs 页面即可出现寄存器页。在进入调试模式前，工程窗口的寄存器页是空白的，进入调试模式后，工程窗口会显示当前模拟状态下单片机寄存器的值，如图 2-23 所示。




Register	Value
Regs	
r0	0x00
r1	0x00
r2	0x00
r3	0x02
r4	0x03
r5	0xf3
r6	0x00
r7	0x01
Sys	
a	0x00
b	0x00
sp	0x7a
sp_max	0x7a
dptr	0x0000
PC \$	C:0x0EFA
status	407
sec	0.00020350
psw	0x00

图 2-23 工程窗口寄存器页

图 2-23 是工程窗口寄存器页的内容，寄存器页包括了当前的工作寄存器组和系统寄存器，系统寄存器组有一些是实际存在的寄存器，如 A、B、DPTR、SP、PSW 等，有一些寄存器是实际中并不存在或虽然存在却不能对其操作的，如 PC、Status 等。每当程序中执行到对某寄存器的操作时，该寄存器会以反色（蓝底白字）显示，用鼠标单击，然后按下 F2 键，即可修改该值。

（2）存储器窗口（Memory Window）

存储器窗口可以直接监控系统中各种内存中的数值，一共有四个页面，即 Memory #1～Memory #4，能够同时监看四个不同区段或种类的内存内容。

执行 View/Memory Window 菜单指令，或者直接点选工具栏中的图标，即可开启 Memory Window。若要查询某段内存内容，必须在 Address 编辑框后输入欲查寻地址范围（“字母：数字”），其中字母可以是 C、D、I、X，分别表示代码存储空间、直接寻址的

片内存储空间、间接寻址的片内存储空间、扩展的外部 RAM 空间，数字代表想要查看的地址。例如，选择 Memory #1，并在 Address 输入“C: 0”，显示从 0000H~FFFFH 地址范围的 ROM 程序内存，即查看程序的二进制代码；选择 Memory #2，并在 Address 输入“D: 0”，显示从 00H~FFH 地址范围的 RAM 数据存储器，如图 2-24 所示。

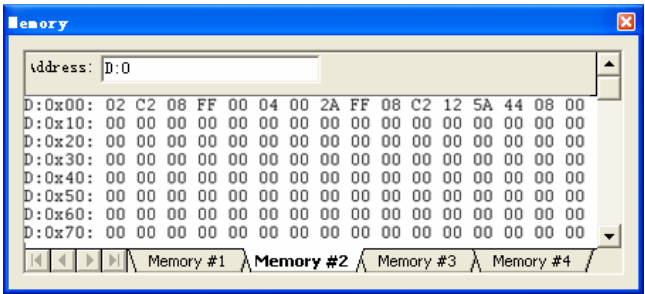


图 2-24 存储器窗口

该窗口的显示值可以以各种形式显示，如十进制、十六进制、Char 型等，改变显示方式的方法是单击鼠标右键，在弹出的快捷菜单中选择，如图 2-25 所示。

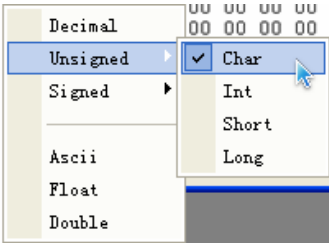



图 2-25 右键快捷菜单

该菜单用分隔条分成两部分，其中第一部分与第二部分的三个选项为同一级别，选中第一部分的任一选项，内容将以整数形式显示。第一部分又有多个选择项，其中 Decimal 项是一个开关，如果选中该项，则窗口中的值将以十进制的方式显示，否则按默认的十六进制方式显示。Unsigned 和 Signed 后分别有 Char、Int、Short、Long 四个选项，分别表示以单字节方式显示、将相邻双字节组成整型数方式显示、将相邻 4B 组成短/长整型方式显示，而 Unsigned 和 Signed 则分别代表无符号形式和有符号形式。究竟从哪一个单元开始的相邻单元则与设置有关，以整型为例，如果输入的是 I:0，那么 00H 和 01H 单元的内容将会组成一个整型数，而如果输入的是 I:1，01H 和 02H 单元的内容组成一个整型数。

选中第二部分的 Ascii 项，则将以字符型显示；选中 Float 项将相邻 4B 组成的浮点数形式显示、选中 Double 项，则将相邻 8B 组成双精度形式显示。

(3) 观察窗口（Watch & Call Stack Windows）

观察窗口是很重要的一个窗口，工程窗口中仅可以观察到工作寄存器和有限的寄存器（如 A、B、DPTR 等），如果需要观察其他寄存器的值，或者在高级语言编程时需要直接观察变量，就要借助于观察窗口。

执行 View/Watch & Call Stack Window 菜单指令，或者直接单击工具栏中的  图标，可以开启 Watch & Call Stack Window，它可以监控程序当中所用变量之值，Locals 页面会自

动列出当前正在执行的函数式之中各变量之值，另外，如果想自定义监看的变量，也可以在 Watch #1/#2 中定义，甚至能够直接变更这些变量的数值（双击变量值），最后 Call Stack 页面列出程序中各函数式呼叫的情形，如图 2-26 所示。

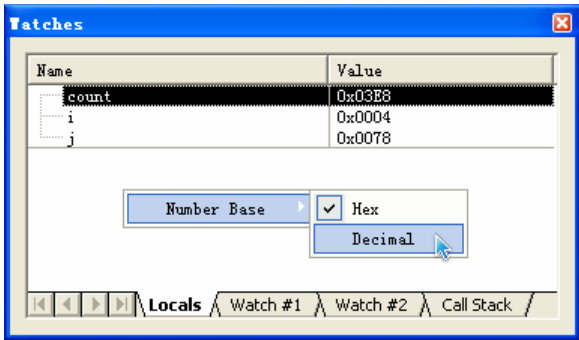


图 2-26 观察窗口

该窗口显示变量值可以以十六进制（Hex）显示，或者十进制（Decimal）显示。在观察窗口单击鼠标右键，执行 Number Base/Decimal 选项命令，改变后的变量数值显示如图 2-27 所示。

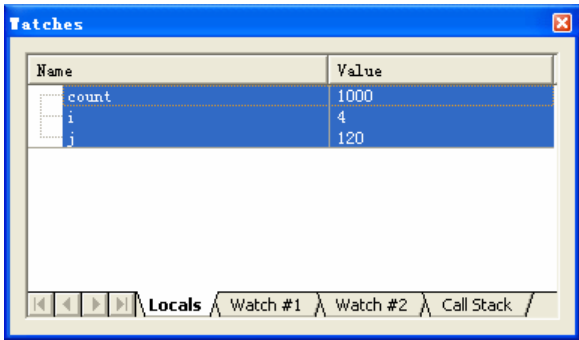



图 2-27 十进制数值显示

（4）反汇编窗口（Disassembly Window）

虽然单芯片程序可由高级语言编写，如 C 语言，但是仍有必要观察程序代码是如何在微处理器内部运作的，反汇编窗口会将已加载机器码程序再转译成基本的汇编语言程序，并标出程序代码在内存的地址供参考。

执行 View/Disassembly Window 菜单命令，或直接单击工具栏中的  图标，打开反汇编窗口，如图 2-28 所示。

（5）串行窗口（Serial Window）

Keil 提供了串行窗口，可以直接在串行窗口中键入字符，该字符虽不会被显示出来，但却能传递到仿真 CPU 中。如果仿真 CPU 通过串行口发送字符，那么这些字符会在串行窗口显示出来，用该窗口可以在没有硬件的情况下用键盘模拟串口通信。

由于部分 CPU 具有双串口，故 Keil 提供了多达 3 个串行窗口，这里选用的 89C51 芯片只有一个串行口，所以 Serial window #2/#3 串行窗口不起作用。

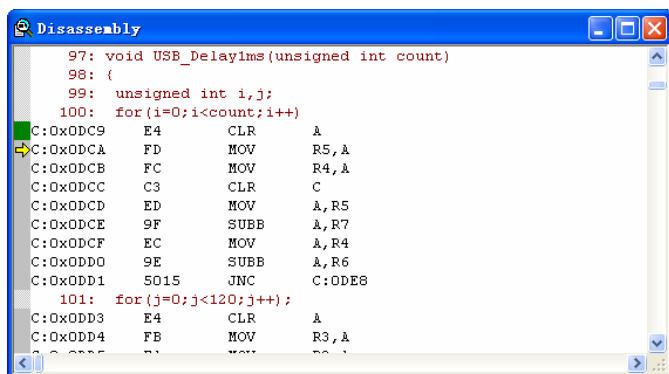


图 2-28 反汇编窗口

一般情况下，仅在单步执行时才对变量的值的变化感兴趣，全速运行时，变量的值是不变的，只有在程序停下来之后，才会将这些值最新的变化反映出来，但是，在一些特殊场合下也可能需要在全速运行时观察变量的变化，此时可以启动周期更新窗口，执行 View/Periodic Window Update 菜单命令，确认该项处于被选中状态，即可在全速运行时动态地观察有关值的变化。但是，选中该项，将会使程序模拟执行的速度变慢。

2.3 汇编语言与C语言的混合使用

对于目前普遍使用的 RISC 架构的 8 位 MCU 来说，其内部 ROM、RAM、Stack 等资源都非常有限。如果使用 C 语言编写，一条 C 语言指令编译后，会变成很多条机器码，很容易出现 ROM 空间不够、堆栈溢出等问题。而且一些单片机厂家也不一定能提供 C 编译器。而汇编语言，一条指令就对应一个机器码，每一步执行的动作都很清楚，并且程序大小和堆栈调用情况都容易控制，调试起来也比较方便。在大多数情况下，汇编语言程序能和 C 语言编写的程序很好地结合起来使用。

2.3.1 汇编语言与C语言的比较

汇编语言是一种用文字助记符来表示机器指令的符号语言，是最接近机器码的一种语言，其主要优点是占用资源少、程序执行效率高。但是不同的 CPU，其汇编语言可能有所不同，所以不易移植。

作为高级语言的 C 语言具有可读性强、编程简单和调试方便的特点。C 语言是目前非常流行的一种编程语言，除具有高级语言使用方便灵活、数据处理能力强、编程简单等优点外，还可实现汇编语言的大部分功能，例如，可直接对硬件进行操作、生成的目标代码质量较高且执行的速度较快等，所以在工程上对硬件处理速度要求不很高的情况下，基本可以用 C 语言代替汇编语言，编写接口电路的控制软件。但 C 语言也不能完全取代汇编语言，例如，在一些对速度要求很高的实时控制系统中，以及对硬件的特殊控制方面，C 语言有时也不能完全很好胜任，还需要用汇编语言来编写，因为汇编语言目标代码更为精练，对硬件直接控制能力更强和执行速度更快，但汇编语言编程烦难、表达能力差。

任何一种语言最终生成的都是机器码，所以生成的机器码可以通过反汇编生成汇编码

（因为汇编语言和机器码是一一对应的，可以通过汇编码生成机器码，也可以通过机器码生成汇编码）。

C 语言和汇编调用函数的过程是一样的，都是压栈、转跳、执行完毕、出栈。

下面是 C 语言和汇编代码的比较，代码所表示的意义是相同的。

C 语言编写的代码如下所示：

```
main()
{
    int a,b,c;
    scanf("%d",&a);
    printf("\n");
    scanf("%d",&b);
    printf("\n");
    scanf("%d",&c);
    printf("\n");
    if(a==b && a==c)
    {
        printf("All equal!");
    }
    else if(a!=b && a!=c && b!=c)
    {
        printf("No equal!");
    }
    else
    {
        printf("Only two equal!");
    }
}
```

汇编语言编写的代码如下所示：

```
dseg segment
msg1 db 'both equal!',0dh,0ah,'$'
msg2 db 'no equal!',0dh,0ah,'$'
msg3 db 'all equal!',0dh,0ah,'$'
ctrlf db 0dh,0ah,'$'
num1 dw ?
num2 dw ?
num3 dw ?
dseg ends
cseg segment
assume cs:cseg,ds:dseg
Start:
```

```

mov ax,dseg
mov ds,ax
mov ah,01h    ; 输入 1 个数
int 21h
mov num1,ax
lea dx,ctrlf
mov ah,09h
int 21h
mov ah,01h    ; 输入第 2 个数
int 21h
mov num2,ax
lea dx,ctrlf
mov ah,09h
int 21h
mov ah,01h    ; 输入第 3 个数
int 21h
mov num3,ax
lea dx,ctrlf
mov ah,09h
int 21h
mov ax,num1
cmp ax,num2
je already_two
mov ax,num2
cmp ax,num3
je two
lea dx,msg2    ; 只有 1 个数相等
jmp done
already_two:    ; 已经 2 个数相等
cmp ax,num3    ; 判断另外两个数
lea dx,msg3    ; 如果相等则三个数都相等
jmp done
two:           ; 2 个数相等
lea dx,msg1
done:
mov ah,9
int 21h
exit:
mov ah,4ch
int 21h
cseg ends
end Start

```


在汇编语言中，要考虑的东西比较多，而在 C 语言中，它封装好了函数，可以方便使用，编译器负责与底层打交道，用户不用直接与底层打交道了。

2.3.2 C语言中嵌入汇编语言

在单片机编程过程中，比较好的解决办法是 C 语言与汇编语言混合编程，即用 C 语言编写软件的调试程序、用户界面以及速度要求不高的控制部分，而用汇编语言对速度敏感部分提供最高速度的处理模块，供 C 语言程序调用。这种方法提供了最佳的软件设计方案，做到了兼顾速度效率高和灵活方便。

1. 嵌入方式

在 C 语言程序中要可以以如下方式加入汇编代码：

```
#pragma ASM
; Assembler Code Here
#pragma ENDASM
```

2. 系统设置

在 Project 窗口中包含汇编代码的 C 语言文件上右键，选择 Options for File ‘main.c’ 选项，如图 2-29 所示。

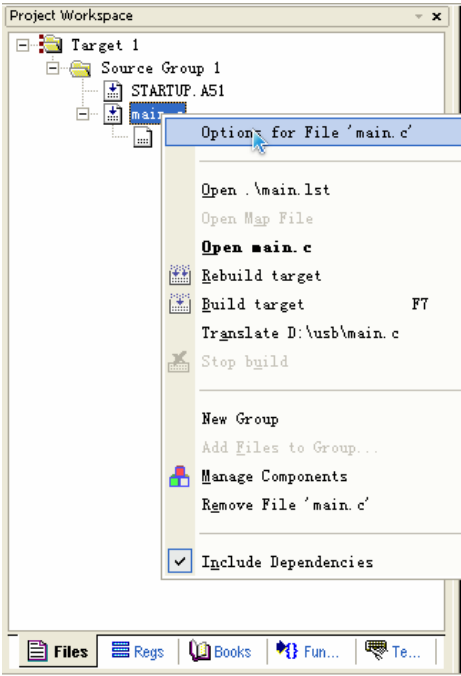


图 2-29 选择 Options for File ‘main.c’ 选项

系统弹出如图 2-30 所示的 Options for File ‘main.c’ 对话框，单击右边的 Generate Assembler SRC File 和 Assemble SRC File，使检查框由灰色变成黑色（有效）状态。

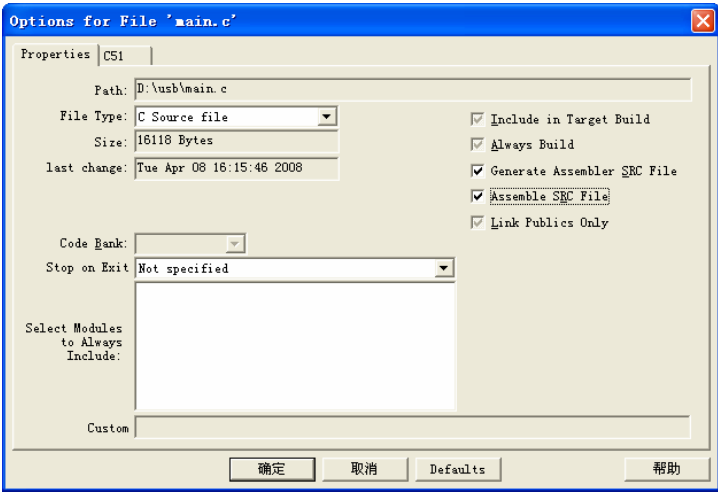


图 2-30 设置 Options for File ‘main.c’ 对话框

根据选择的编译模式，把相应的库文件加入工程中，例如，在 Small 模式时，库文件是 Keil\C51\Lib\C51S.lib，该文件必须作为工程的最后文件。

接下对程序代码进行编译，即可生成目标代码。

下面是 C 语言程序调用汇编语言代码实例。

```
#include <reg51.h>
void main(void)
{
    P2=1;
    #pragma asm
        MOV R7,#10
    DEL:MOV R6,#20
        DJNZ R6,$
        DJNZ R7,DEL
    #pragma endasm
    P2=0;
}
```

3. 无参数传递的函数调用

首先看一个例子，其中 example.c 和 example.a51 为项目中的两个文件。

```
*****example.c*****
extern void delay100();
main()
{delay100;}

*****example.a51*****
?PR?DELAY100 SEGMENT CODE;           //在程序存储区中定义段
PUBLIC DELAY100;                     //声明函数
```

```
RSEG ?PR?DELAY100;                                //函数可被链接器放在任何地方
DELAY100:

    MOV R7, #10

DEL:

    MOV  R6, #20
    DJNZ R6, $
    DJNZ R7, DEL
    RET
    END
```

在 example.c 文件中，先声明外部函数，然后直接在 main()中调用即可。

在 example.a51 中，“?PR?DELAY100 SEGMENT CODE;” 语句的作用是在程序存储区中定义段，“DELAY100”为段名，“?PR?”表示段位于程序存储区内。“PUBLIC DELAY100;”语句的作用是声明函数为公共函数。“RSEG ?PR?DELAY100;”语句表示函数可被链接器放在任何地方，“RSEG”是段名的属性，段名的开头为“PR”，是为了和 C51 内部命名转换兼容，命名转换规律如表 2-1 所示。

表 2-1 内存空间与命名转换对应表

内存空间	命名转换
CODE	?PR?, ?CD
XDATA	?XD
DATA	?DT
BIT	?BI
PDATA	?PD

值得注意的是，建立的 C 文件和 A51 文件不能使用同一个文件名。

4. 有参数传递的函数调用

在 C51 和汇编之间传递参数的方式是通过寄存器传递参数，C51 中不同类型的实参会存入相应的寄存器，在汇编中只需对相应寄存器进行操作，即可达到传递参数的目的。

不同类型的数据及其传递参数的寄存器如下所示。

在 C 语言和汇编语言混合编程的时候，存在 C 语言和汇编语言的变量以及函数的接口问题。在 C 语言程序中定义的变量，编译为.asm 文件后，都被放进了.bss 区，而且变量名的前面都带了一个下划线。在 C 语言程序中定义的函数，编译后在函数名前也带了一个下划线。

例如：

```
extern int num 就会变成 .bss _num, 1
extern float nums[5]就会变成.bss _nums, 5
extern void func ( )就会变成 _func,
```

汇编语言和 C 语言的相互调用可以分以下几种情况：

(1) 汇编语言程序中访问 C 语言程序中的变量和函数

在汇编语言程序中，用 `_XX` 就可以访问 C 语言程序中的变量 `XX` 了。访问数组时，可以用 `_XX+偏移量` 来访问，如 `_XX+3` 访问了数组中的 `XX[3]`。

在汇编语言程序调用 C 语言函数时，如果没有参数传递，直接用 `_funcname` 就可以了。如果有参数传递，则函数中最左边的一个参数由寄存器 `A` 给出，其他参数按顺序由堆栈给出。返回值是返回到 `A` 寄存器或者由 `A` 寄存器给出的地址。同时注意，为了能够让汇编语言程序能访问到 C 语言程序中定义的变量和函数，它们必须声明为外部变量，即加前缀 `extern`。

(2) C 语言程序中访问汇编语言程序中的变量

如果需要在 C 语言程序中访问汇编语言程序中的变量，则汇编语言程序中的变量名必须以下划线为首字符，并用 `global` 使之成为全局变量。

如果需要在 C 语言程序中调用汇编语言程序中的过程，则过程名必须以下划线为首字符，并且，要根据 C 语言程序编译时使用的模式是 `stack-based model` 还是 `register argument model` 来正确地编写该过程，使之能正确地取得调用参数。

(3) 在线汇编

在 C 语言程序中直接插入 `asm(“***”)`，内嵌汇编语句，需要注意的是这种用法要慎用，在线汇编提供了能直接读/写硬件的能力，例如，读/写中断控制允许寄存器等，但编译器并不检查和分析在线汇编语言，插入在线汇编语言改变汇编环境或可能改变 C 语言程序中变量的值，从而可能导致严重的错误。

2.3.3 汇编语言程序调用C语言程序

下面介绍如何在汇编语句中调用 C 语言程序代码。

首先看一下如下的程序代码。

```
int g(int a, int b, int c, int d, int e)
{
    return a + b + c + d + e;
}
```

汇编语言程序调用 C 语言程序 `g()` 计算 5 个整数 i , $2 \times i$, $3 \times i$, $4 \times i$, $5 \times i$ 的和。

```
EXPORT f
AREA f, CODE, READONLY
IMPORT g                ; 使用伪操作数 IMPORT 声明 C 程序 g()
STR lr, [sp, #-4]!      ; 保存返回地址
ADD r1, r0, r0           ; 假设进入程序 f 时，r0 中的值为 i，r1 值设为  $2 \times i$ 
ADD r2, r1, r0           ; r2 的值设为  $3 \times i$ 
ADD r3, r1, r2           ; r3 的值设为  $5 \times i$ 
STR r3, [sp, # -4]!     ; 第五个参数  $5 \times i$  通过数据栈传递
ADD r3, r1, r1           ; r4 值设为  $4 \times i$ 
BL g                    ; 调用 C 程序 g()
ADD sp, sp, #4          ; 调整数据栈指针，准备返回
```

```
LDR    pc, [sp], #4      ; 返回
END
```

值得注意的是汇编语言程序在编写时一定要缩进，否则编译将会出现错误。

需要注意的是，在 C 语言中，对于局部变量的建立和访问，是通过堆栈实现的，它的寻址是通过堆栈寄存器 SP 实现的。而在汇编语言中，为了使程序代码变得更为精简，T1 在直接寻址方式中，地址的低 7 位直接包含在指令中，这低 7 位所能寻址的具体位置可由 DP 寄存器或 SP 寄存器决定。具体实现可通过设置 ST1 寄存器的 CPL 位实现，CPL=0，DP 寻址；CPL=1，SP 寻址。在 DP 寻址的时候，由 DP 提供高 9 位地址，与低 7 位组成 16 位地址；在 SP 寻址的时候，16 位地址是由 SP（16 位）与低 7 位直接相加得来。

由于在 C 语言的环境下，局部变量的寻址必须通过 SP 寄存器实现，在混合编程的时候，为了使汇编语言不影响堆栈寄存器 SP，通常的方式是在汇编环境中使用 DP 方式寻址，这样可以使二者互不干扰。编程中只要注意对 CPL 位正确设置即可。

在汇编语言中，.word 的意思相当于 C 语言里的 int、char 等定义一个变量的范围。

程序编译过程中，出现编译错误原因有 2 个：

- 如果在汇编语言里面定义.global（全局符号），那么在 C 语言里面应该用 extern 声明，以引用该符号。
- 在汇编语言里面声明的时候，符号前应加下划线，如“FIQ_Addr: .word EXTint_FIQ”应为“FIQ_Addr: .word _EXTint_FIQ”。在 C 语言里面应用 extern 声明。另外的一种方法是用.ref 代替.global 来声明符号，这样就不用用在 C 语言源程序里面用 extern 声明了。两种方法结果相同。这里指的是用 C 语言和汇编语言混编程用法，至于 C++语言变量如何翻译成汇编语言符号可以用仿真器，原则类似。

汇编语言与 C 语言混合编程的关键问题有如下几点。

1. C语言程序变量与汇编语言程序变量的共用

为了使程序更易于接口和维护，可以在汇编语言程序中引用与 C 语言程序共享的变量：

.ref_to_dce_num,_to_dte_num,_to_dce_buff,_to_dte_buff

在汇编语言程序中引用而在 C 语言程序可直接定义的变量：

```
unsigned char to_dte_buff[BUFF_SIZE];    //单片机发向 PC 的数据
int to_dte_num;                          //缓冲区中存放的有效字节数
int to_dte_store;                        //缓冲区的存放指针
int to_dte_read;                         //缓冲区的读取指针
```

这样经过链接就可以完成对应。

2. 程序入口问题

在 C 语言程序中，程序的入口是 main()函数。而在汇编语言程序中其入口由*.cmd 文件中的命令决定，例如，-e main_start；程序入口地址为 main_start。这样，混合汇编出来的程序得不到正确结果。因为 C 语言到 ASM 的汇编有默认的入口 c-int00，从这开始的一段程序为 C 语言程序的运行做准备工作。这些工作包括初始化变量、设置栈指针等，系统不能跨越。这时可在*.cmd 文件中去掉语句-e main_start。如果仍然想执行某些汇编语言程序，可以

C 语言函数的形式执行。

例如：

```
main_start(); //其中含有其他汇编语言程序
```

但前提是在汇编语言程序中把_main_start 作为首地址，程序以 rete 结尾（作为可调用的函数）的程序段，并在汇编语言程序中引用_main_start，即.ref _main_start。

3. 移位问题

在 C 语言中把变量设为 Char 型时，它是 8 位的，但在单片机汇编语言中此变量仍被作为 16 位处理，所以会出现在 C 语言程序中的移位结果与汇编语言程序移位结果不同的问题。解决的办法是在 C 语言程序中，把移位结果再用 0x00ff 去进行与操作即可。

4. 堆栈问题

汇编语言程序对堆栈的依赖很小，但在 C 语言程序中分配局部变量、变量初始化、传递函数变量、保存函数返回地址、保护临时结果功能都是靠堆栈来完成的。而 C 编译器无法检查程序运行时堆栈能否溢出。

5. 程序跑飞问题

编译后的 C 语言程序跑飞一般是由于对不存在的存储区访问而造成的。

首先要查.MAP 文件与 memory map 对比，看是否超出范围。如果有中断的程序中跑飞，应重点查在中断程序中是否对所用到的寄存器进行了压栈保护。如果在中断程序中调用了 C 语言程序，则要查汇编后的 C 语言程序中是否用到了没有被保护的寄存器并提供保护（在 C 语言程序的编译中是不对 A、B 等寄存器进行保护的）。

第3章 RTX51 实时操作系统

通用计算机具有完善的操作系统（OS）和应用程序接口（API），它们是计算机基本组成不可缺少的一部分，应用程序的开发以及完成后的软件都在 OS 平台上面运行，但通常都不是实时的。嵌入式系统则不同，应用程序可以没有操作系统直接在芯片上运行，需要选配嵌入式实时操作系统（RTOS）开发平台，这样才能保证程序执行的实时性、可靠性，并减少开发时间，保障软件质量。

3.1 RTX51 操作系统简介

RTX51 是一个小型的实时操作系统，在 C51 单片机开发中常会用到此系统。RTX51 操作系统适用于 8051 系列单片机的实时操作系统。RTX51 可以使复杂的系统和软件设计以及有时间限制的工程开发变得简单。RTX51 是一个强大的工具，它可以在单个 CPU 上管理多个任务。本章将介绍 RTX51 实时操作系统的功能，以及如何使用 RTX51 实时操作系统。

3.1.1 实时操作系统（RTOS）

实时操作系统（real time operating system，RTOS）是嵌入式应用软件的基础和开发平台。实时是指物理进程的真实时间，实时操作系统是指具有实时性，能支持实时控制系统工作的操作系统。RTOS 是针对不同处理器优化设计的高效率实时多任务内核，将 CPU 时间、中断、I/O、定时器等资源都包装起来，留给用户一个标准的 API，并根据各个任务的优先级，合理地在不同任务之间分配 CPU 时间。

RTOS 最关键的部分是实时多任务内核，它的基本功能包括任务管理（多任务和基于优先级的任务调度）、定时器管理、存储器优化管理（含 ROM 的管理）、资源管理、事件管理、系统管理、消息管理、队列管理、旗语管理等，这些管理功能是通过内核服务函数的形式交给用户调用的，也就是 RTOS 的 API。

实时操作系统中的任务（task）相当于分时操作系统中的进程（process）的概念。系统中的任务有运行（executing）、就绪（ready）、挂起（suspended）、冬眠（dormant）四种状态。

- ① 运行，获得 CPU 控制权；
- ② 就绪，进入任务等待队列，通过调度转为运行状态；
- ③ 挂起，任务发生阻塞，移出任务等待队列，等待系统实时事件的发生而唤醒，从而转为就绪或运行；
- ④ 冬眠，任务完成或错误等原因被清除的任务，也可以认为是系统中不存在了的任务。系统中只能有一个任务处于运行状态，各任务按级别通过时间片分别获得对 CPU 的访问权。

3.1.2 RTX51 实时操作系统

RTX51 是一个小型的实时操作系统，在 C51 单片机开发中常会用到此系统。RTX51 操作系统适用于 8051 系列单片机的实时操作系统。RTX51 可使复杂的系统和软件设计以及有

时间限制的工程开发变得简单。RTX51 主要有两个不同的可用版本，分别为 RTX51 Tiny 和 RTX51 Full。

RTX51 Full 允许 4 个优先权任务的循环和切换，并且还能并行的利用中断功能。RTX51 支持信号传递，以及与系统邮箱和信号量进行消息传递。RTX51 的 os_wait 函数可以等待以下事件：中断、时间到、来自任务或中断的信号、来自任务或中断的消息、信号量。

RTX51 Tiny 版本是 RTX51 Full 的一个子集，它可以运行在 8051 的单芯片嵌入式系统上，并且不需要任何外部数据存储器，但也不排斥应用程序访问外部的数据存储器。RTX51 Tiny 版本可以使用 C51 编译器所支持的所有存储模式。所使用的存储模式只影响应用对象的存储位置。RTX51 Tiny 的系统变量和应用程序的堆栈区总是存储在 8051 单片机的片内 RAM 中（即 DATA 和 IDATA）。典型的 RTX51 Tiny 应用程序一般运行于 Small 存储模式下。RTX51 Tiny 允许循环任务切换，并且支持信号传递，还能并行地利用中断功能。

RTX51 Tiny 版本使用了 8051 的定时器 0 和定时器 1 的中断信号。SFR 中的全局中断允许位或定时器 0 中断屏蔽位都可能使 RTX51 Tiny 停止运行。因此，除非有特殊的应用目的，应该使定时器 0 的中断始终开启，以保证 RTX51 Tiny 的正常运行。

由于 Keil C 中自带了 RTX51 Tiny，事实上 RTX51 Tiny 也能够满足绝大部分场合的应用要求，所以本书只讲解 RTX51 Tiny 的内容，出现 RTX51 的地方默认为 RTX51 Tiny。

RTX51 可以在所有的 8051 系列单片机上运行。用户只需要用标准的 C 语言编写 RTX51 程序，然后用 C51 编译器编译即可生成代码。其中，仅有少数内容和标准 C 语言有差异，这些内容是为了实现任务标识和优先级而设置的。RTX51 程序设计需要包含实时运行头文件和必要的库文件，并且要用 BL51 链接/定位器来实现链接。在 Keil 中，只需要在目标选项的 Target 标签中的 Operating 中选择“RTX51 Tiny”，在头文件中加上#include <rtx51tiny.h>即可。

1. 性能参数

实时操作系统的性能参数对嵌入式系统的应用开发也有着直接影响，RTX51 的性能参数如表 3-1 所示。

表 3-1 RTX51 的性能参数

描 述	RTX51 Tiny 版本
任务数	16
RAM 需求	7 B DATA 3X（任务数）B IDATA
代码要求	900 B
硬件要求	定时器 0
系统时钟	100~65 535 周期
中断响应时间	<20 周期
任务切换时间	100~700 周期，依赖于堆栈装载

2. RTX51 的程序结构

RTX51 Tiny 通过循环（round-robin）方式来实现多任务，以达到多个无限循环或任务

的准并行执行。这里的多任务并不是真正同时执行的，而是使用不同的时间片来执行，即只是宏观上的同时执行。它将可用的 CPU 周期分成多个时间片，由 RTX51 把这些时间片分配给每一个任务使用。每个任务只能在预定的时间片里运行。然后，RTX51 再切换到另一个已经准备就绪的任务，让它再执行一定的时间片。

时间片一般是比较短的，一个时间片大约只有毫秒级时间。正是由于这个原因，在用户看来，多个任务似乎是在同时执行的。

RTX51 利用了一个由 8051 定时器中断信号驱动的定时程序来实现控制。定时器产生的周期性中断信号用来驱动 RTX51 的定时节拍。

RTX51 与用户程序中的 main 函数是无关的，用户程序中即使没有 main 程序，操作系统也会自动从设定的任务 0 开始执行。如果用户程序中已经有了 main 函数，就必须用人工方式来启动 RTX51，对于 RTX51 Tiny 而言，可以调用 os_create_task 函数来完成。

下面是一个只使用 round-robin 任务方式的简单 RTX51 程序实例。程序中的两个任务都是简单的计数循环。RTX51 从 book0 函数（设定为任务 0）开始执行。程序中还有另一个名为 book1 的任务（设定为任务 1）。系统一旦启动，在 book0 执行一段时间后，RTx51 就自动切换到 book1 的执行；而 book1 执行一段时间后，RTX51 又切换回 book0……如此无限循环地重复执行下去。

程序代码如下所示：

```
#include <rtx51tiny.h>

int  counter0;
int  counter1;
void  book0 (void) _task_ 0
{
    os_create_task(1);           //任务 1 已准备就绪
    while (1)                   //无限循环
    {
        counter0++;             //更新计数值 counter0
    }
}
void  book1 (void) _task_ 1
{
    while(1)                    //无限循环
    {
        counter1++;             //更新计数值 counter1
    }
}
```

3. RTX51 程序的控制机制

os_wait 函数提供了一种更为有效的方式，来给几个任务分配可使用的处理器时间。os_wait 函数中断当前正在运行的任务，并且等待特定的事件。在一个任务等待事件的时间里，其他任务可以被执行。

可以用 `os_wait` 函数等待的最简单的事件是事件溢出和 RTX51 时钟报时信号周期，这类事件可被用于任务中需要延迟一段时间的地方。这可用于查询一个开关状态的代码中，在这样的条件下，只需每隔 50ms 左右查询一次开关。

RTX51 使用 8051 单片机的一个定时器来产生一个循环的中断（时钟周期）。响应 `os_wait` 的最简单事件是时间到，当前正在执行的任务被指定的时钟周期所中断。

下面的程序可以在允许其他的任务执行的时候使用 `os_wait` 功能延迟任务的执行。程序代码如下：

```
#include <rtx51tny.h>
int counter0;
int counter1;
void book0 (void)_task_0
{
    os_create(1);                //任务 1 已准备就绪
    while(1)                    //无限循环
    {
        counter0++;             //counter0 递增
        os_wait(K_TMO,4);       //暂停 4 个定时节拍
    }
}
void book1 (void)_task_1
{
    while(1)                    //无限循环
    {
        counter1++;             //counter1 递增
        os_wait(K_TMO,3);       //等待一个信号
    }
}
```

在上面的程序中，首先启动 `book0`，然后在增加 `counter0` 计数以后，`job0` 调用 `os_wait` 函数，RTX51 等待 4 个时钟周期，直到 `book0` 准备好再次运行为止。在这期间，RTX51 切换到下一个任务 `book1`。`book1` 也调用了 `os_wait` 函数，等待 3 个时钟周期。现在 RTX51 没有其他任务需要执行，因此在它可以延续执行 `book0` 之前进入一个空循环，等待各时钟报时信号过去。

本例子的结果是 `counter0` 每 4 个时钟报时周期加 1，而 `counter1` 每 3 个时钟报时周期加 1。

`os_wait` 函数的另一个事件是信号，被用来协调任务，直到另一个任务发出信号，在 `os_wait` 函数控制下的任务才结束等待状态。如果信号预先就被发送出来，那么任务将立即继续执行。

以下例子说明了这种应用。

```
#include <rtx51tny.h>
int counter0;
int counter1;
```

```

void book0 (void)_task_0
{
    os_create(1);                //任务 1 已准备就绪
    while(1)                    //无限循环
    {
        if(++counter0==0)        //counter0 递增
            os_send_signal(1);
    }
}

void book(void)_task_1
{
    while(1)                    //无限循环
    {
        os_wait(K_SIG,0,0);      //等待一个信号
        counter1++;              //counter1 递增
    }
}

```

在上述例子中，book1 一直处于等待状态，直到它接收到从任何其他任务发出的信号。当它接收到一个信号时，它将使 counter1 加 1，然后继续等待另一个信号。book0 将连续地增加 counter0，直到它溢出到 0。当溢出发生时，book0 发送一个信号给 book1，同时 RTX51 标记 book1 为执行状态。在 RTX51 到达下一个时钟报时周期前，book1 不会开始执行。

在上一个例子中，book1 收到一个信号后不会立即开始，只有当 book0 发生了时间到事件后，book1 才会启动。如果 book1 被赋予了比 book0 高的优先级，通过抢先任务切换；如果 book1 收到了信号，就会立即开始。优先级在任务定义中被指定（默认的优先级是 0）。

RTX51 允许指定任务的优先级。RTX51 Full 提供了抢先的任务切换，RTX51 Tiny 不具备这个功能。一个具有较高优先级的任务变成可用的时，会中断一个低优先级任务或抢在它前面执行，这叫做优先型多任务或仅仅称为抢先机制。

可以变更上述函数 book1 的说明，给它一个比 book0 更高的优先级。全部任务的默认优先级均为 0，这是最低的优先级，优先级可以设定为 0~3。

下面的例子说明如何定义 book1 的优先级为 1。

程序代码如下：

```

void book1 (void)_task_1_priority_1          //设置优先级为 1
{
    while (1)
    {
        os_wait (K_SIG, 0, 0);                //等待一个信号
        counter1++;                            //counter1 递增
    }
}

```

现在每当 book0 发送一个信号给 book1 时，book1 将立即开始执行。

4. RTX51 的任务调度

RTX51 是完全地统一到 C51 程序设计语言中的，上述的例子是可执行的 RTX51 程序，不需要书写任何 8051 汇编语言程序或函数，唯一需要做的是用 C51 编译 RTX51 程序，并把它们用 BL51 Linker/Locator 链接在一起。

例如，如果使用 RTX51 Tiny，可以使用以下命令行命令进行编译和链接：

```
C51 EXAMPLE.C
BL51 EXAMPLE.OBJ RTX51TINY
```

如果使用 RTX51，可以使用以下命令行命令进行编译和链接：

```
C51 EXAMPLE.C
BL51 EXAMPLE.OBJ RTX51
```

RTX51 利用任务状态来管理各个任务。用户为 RTX51 定义的每个任务都会以各种状态的某一种来运行。RTX51 内核为每个任务保留了适当的状态，如表 3-2 所示。

表 3-2 状态描述说明

状 态	描 述
running 运行状态	当前正在执行的任务，在任一时刻只能由一个任务处在运行状态
ready 就绪状态	等待执行的任务，当前任务执行完后，接着执行就绪任务
waiting 等待状态	等待某一事件的任务，若事件发生，任务进入就绪状态
deleted 删除状态	没有启动的任务
time-out 超时状态	与就绪状态相似，放在 Round-Robin 循环中尚未执行的任务

RTX51 的等待功能支持以下事件：

- 超时（timeout）：挂起运转的任务指定数量的时钟报时周期；
- 间隔（interval）：仅在 RTX51 Tiny 中使用，类似于超时，但是软件定时器没有复位来产生循环的间隔（时钟所需要的）；
- 信号（signal）：用于任务内部协调；
- 信息（message）：仅适用于 RTX51 Full，用于信息的交换；
- 中断（interrupt）：仅适用于 RTX51 Full，一个任务可以等待 8051 硬件中断；
- 旗标标志（semaphore）：仅适用于 RTX51 Full，旗标标志用于管理共享的系统资源。

RTX51 以 Round-Robin 多任务方式执行程序，它支持多个无限循环或任务的准并行执行。任务不是被同时执行，而是以分时的方式轮片执行。可用的 CPU 时钟周期被分成多个时间片，然后由 RTX51 将这些时间片分配给各个任务。每个任务只允许在预定的时间片中执行，时间片用完时，RTX51 就切换至另一个已经就绪的任务，继续执行一段时间。时间片的具体长度可以用配置函数 timesharing 来定义。

如果遇到因为一个任务处于等待并且占用了时间片而暂时无法往下执行，可以调用系统函数 os_wait 来通知 RTX51，以便将当前的任务挂起而提前执行另一任务。

RTX51 中处理任务分配的模块称为调度程序（scheduler），调度程序驱动哪个任务运行是按照以下的规则进行的：

- ① 如果发生以下情况时，当前运行的任务将被中断：
 - 调用 `os_wait` 函数，而所等待的事件未来到；
 - 任务的执行时间已经超过所定义的 Round-Robin 循环时间间隔。
- ② 如果发生以下情况时，另一个任务将被启动：
 - 已没有正在执行的任务；
 - 将要执行的任务处在就绪状态或超时状态。

5. RTX51 的技术参数

实时操作系统的性能参数对嵌入式系统的应用开发也有着直接影响，RTX51 的性能参数如表 3-3 所示。

表 3-3 RTX51 性能

描 述	RTX51 Full	RTX51 Tiny
任务数量	最多 256 个；可同时激活 19 个	16 个
RAM 需求	40~46B 的 DATA 空间 20~200 B 的 IDATA 空间（用户堆栈） 最小 650 B 的 XDATA 空间	7B 的 DATA 空间 3 倍于任务数量的 IDATA 空间
代码要求	6~8 KB	900 B
硬件要求	定时器 0 或定时器 1	定时器 0
系统时钟	1 000~40 000 个周期	1 000~65 535 周期
中断请求时间	小于 50 个周期	小于 20 个周期
任务切换时间	70~100 个周期（快速任务），180~700 个周期 （标准任务），取决于堆栈的负载	100~700 个周期，取决于堆栈的负载
邮箱系统	8 个分别带有整数入口的信箱	不提供
内存池	最多 16 个内存池	不提供
信号量	8×1 位	不提供

6. RTX51 的系统配置

编写 RTX51 程序需要包含 `rts51tiny.h` 文件。在程序中，需要用关键字 “`_task_`” 来声明一个函数的任务属性。RTX51 程序不需要 `main` 函数。在进行链接处理时，会将启动 `book0` 的执行所需要的代码链接进来，作为开始执行的代码。

用户可以更改配置文件 `conf_tiny.a51` 中的以下几个参数：

- 系统定时器中断所用的寄存器组；
- 系统定时器的时间间隔；
- Round-Robin 的超时（Time-Out）值；
- 内部数据存储器的大小；
- RTX51 Tiny 启动后自由堆栈的大小。

以下是配置文件的部分内容：

- RTX51 的硬件定时器；
- 用下面的 EQU 可预置 RTX51 的定时器时间常数；

- 用 8051 定时器 0 作为控制软件的定时器；
- 定义定时器中断用的寄存器组，例如，
INT_REGBANK EQU 1 ;默认为寄存器 1 组
- 定义 8051 定时器 0 溢出所需的机器周期数，例如，
INT_CLOCK EQU 10000 ;默认周期数为 10 000
- 定义 Round-Robin 的 Timeout 所需的定时器溢出数，例如，
TIMESHARING EQU 5 ;默认为 5 次

注意：Round-Robin 任务切换可用 timesharing 为 0 来屏蔽

- RTX51 堆栈空间；

以下的 EQU 语句定义了堆栈区的片内 RAM 体积和最小自由堆栈空间：

- 定义了堆栈空间耗尽后所执行的宏代码；
- 定义最大的堆栈 RAM 地址，例如，
RAMTOP EQU 0FFH ;默认地址是 255
- 定义最小的堆栈自由空间，例如，
FREE_STACK EQU 20 ;默认为 20 B 堆栈自由空间；
- 发生堆栈用尽时的执行代码，例如，

```
STACK_ERROR MACRO
CLR EA      ; 关闭所有中断
SJMP $      ; 如堆栈空间耗尽，进入死循环
ENDM
```

在这个配置文件中，定义了许多可以修改的参数，以适应用户特定的应用程序环境。这些参数的说明如表 3-4 所示。

表 3-4 配置文件参数说明

参 数	描 述
INT_REGBANK	说明 RTX51 系统所用的寄存器组
INT_CLOCK	定义系统时间间隔，系统用这个间隔产生一个中断信号；定义的数据是指每次中断发生所需的 CPU 周期数
timesharing	定义 Round-Robin 任务切换的超时间隔（time-out），是定时器溢出中断次数。发生指定次数中断后切换任务。如果是 0，则多任务 Round-Robin 机制被屏蔽
ramtop	说明 8051 片内 RAM 的最大地址，8051 为 7FH，8052 为 0FFH
free_stack	定义任务切换是堆栈自由空间体积字节数；RTX51 会检验堆栈体积是否合理。如太小，引用 stack_error 宏
stack_error	当 RTX51 检测到堆栈出错时执行的宏，可以根据应用程序需求更换这个宏

3.2 软硬件需求与定义

本节将讲述 RTX51 Tiny 的软硬件需求，并定义了在本手册中使用的术语。RTX51 Tiny 使用一些综合的系统调用，如同建立在 C51 编译程序内部的使用 _task_关键字定义任务一

样。本节还将介绍 RTX51 Tiny 的任务定义和重要的特征。

1. 工具需要

使用 RTX51 Tiny 需要以下的软件产品：

- C51 编译器；
- A51 宏汇编器；
- BL51 连接器或 LX51 链接器；
- rtx51tiny.lib 和 rtx51bt.lib 库文件的路径是\Keil\C51\LIB 文件夹，这是系统认定的库路径下；
- rtx51tiny.h 必须保存在包含路径下，通常是\Keil\C51\INC 文件夹。

2. 目标需求

RTX51 Tiny 运行于大多数 8051 单片机及其兼容的器件上。RTX51 Tiny 应用程序可以访问外部数据存储器，但内核无此需求。

RTX51 Tiny 支持 Keil C51 编译器全部的存储模式。存储模式的选择只影响应用程序对象的位置，RTX51 Tiny 系统变量和应用程序栈空间总是位于 8051 单片机的内部存储区（DATA 或 IDATA 区），一般情况下，应用程序应使用小（Small）模式。

RTX51 Tiny 执行协作式任务切换，即每个任务调用一个操作系统例程，以及时间片轮转任务切换，即每个任务在操作系统切换到下一个任务前运行一个固定的时间段。不支持抢先式任务切换以及任务优先级。RTX51 Full 支持抢先式任务切换。

RTX51 Tiny 没有按照 bank switching 程序设计，如果 code banking 应用程序需要实时的多重任务，需要使用 RTX51 Full 实时执行程序。

3. 中断处理

RTX51 Tiny 与中断函数并行运作，中断服务程序可以通过发送信号（使用 isr_send_signal 函数）或设置任务的就序标志（使用 isr_set_redy 函数）与 RTX51 Tiny 的任务进行通信。

如同在一个标准的，没有 RTX51 Tiny 的应用中一样，中断例程必须在 RTX51 Tiny 应用中实现并允许，RTX51 Tiny 没有中断服务程序的管理。

RTX51 Tiny 使用定时器 0、定时器 0 中断和寄存器组 1。如果在程序中使用了定时器 0，则 RTX51 Tiny 将不能正常运行，可以在 RTX51 Tiny 定时器 0 的中断服务程序后追加自己的定时器 0 中断服务程序代码。

RTX51 Tiny 假设总中断总是允许的（EA=1）。RTX51 Tiny 库例程在需要时改变中断系统（EA）的状态，以确保 RTX51 Tiny 的内部结构不会被中断破坏。当允许或禁止总中断时，RTX51 Tiny 只是简单地改变 EA 的状态，不保存和重装 EA，EA 只是简单地被置位或清除。因此，如果程序在调用 RTX51 例程前禁止了中断，RTX51 可能会失去响应。

在程序的临界区，可能需要在短时间内禁止中断。但是，在中断禁止后，不能调用任何 RTX51 Tiny 的例程。如果程序确实需要禁止中断，持续的时间应该很短。

4. 再入函数

C51 编译器提供对再入函数的支持，再入函数在再入堆栈中存储参数和局部变量，从而保护递归调用或并行调用。RTX51 Tiny 不支持对 C51 编译器再入栈的任何管理。因此，如果在程序中使用再入函数，必须确保此函数不调用任何 RTX51 Tiny 系统函数，且不被循环任务切换而打断。

仅用寄存器传递参数和保存自动变量的 C 语言函数具有内在的再入性，可以无限制地调用 RTX51 Tiny。

非再入 C 语言函数不能被超过一个以上的任务或中断过程调用，在静态存储区段保存参数和自动变量（局部数据），该区域在函数被多个任务同时调用或递归调用时可能会被修改。

如果确定多个任务不会被递归（或同时）调用，则多个任务可以调用非再入函数。通常，这意味着必须禁止循环任务调度，且该非再入函数不能调用任何 RTX51 Tiny 系统函数。

如果希望在多个任务或中断中调用再入或非再入函数，应当禁止循环任务调度。

5. C51 库例程

可再入 C51 库函数可在任何任务中无限制地使用，对于非再入的 C51 库函数，同样有非再入 C 语言函数的限制。

6. 多数据指针

Keil C51 编译器允许使用多数据指针（存在于许多 8051 兼容的芯片中），RTX51 Tiny 不提供对它们的支持。因此，在 RTX51 Tiny 的应用程序中应小心使用多数据指针，必须确保循环任务切换不会在执行改变数据指针选择器的代码时发生。如果要使用多数据指针，应该禁止循环任务切换。

7. 运算单元

Keil C51 编译器允许使用运算单元（存在于许多 8051 兼容的芯片中）。RTX51 Tiny 不提供对它们的支持。因此，在 RTX51 Tiny 的应用程序中须小心使用运算单元。必须确保循环任务切换不会在执行用运算单元的代码时发生。如果希望使用运算单元，应禁止循环任务切换。

8. 寄存器组

RTX51 Tiny 分配所有的任务到寄存器 0，因此，所有的函数必须用 C51 编译器的默认设置进行编译，registerbank (0)。

中断函数可以使用剩余的寄存器组，但是 RTX51 Tiny 需要寄存器组区域中的 6 个永久性的字节，用于这些字节的寄存器组在配置文件中指定。

3.3 RTX51 的功能函数

下面列出了针对 RTX51 Full 的一些函数，并带有简要的说明和执行时间，如表 3-5 所示。

表 3-5 RTX51 Full 的功能函数

函 数	描 述	CPU 周期
isr_recv_message ⁺	收到消息（来自中断调用）	71（具有消息）
isr_send_message ⁺	发送消息（来自中断调用）	53
isr_send_signal	给任务发去信号（来自中断调用）	46
os_attach_interrupt ⁺	分配中断资源给任务	119
os_clear_signal	删除一个以前发送的信号	57
os_create_task	将一个任务放入执行队列中	302
os_create_pool ⁺	定义一个内存池	644（20×10 B）
os_delete_task	从执行队列中移走一个任务	172
os_detach_interrupt ⁺	移走一个分配的中断	96
os_disable_isr ⁺	禁止 8051 硬件中断	81
os_enable_isr ⁺	允许 8051 硬件中断	80
os_free_block ⁺	释放一块存储空间给内存池	160
os_get_block ⁺	从内存池获得一块存储空间	148
os_send_message ⁺	发送一条消息（从任务中调用）	443（具有任务切换）
os_send_token ⁺	发送一个信号量（从任务中调用）	343（具有快速任务切换）
		94（没有任务切换）
os_set_slice ⁺	设置 RTX51 系统时钟时间片	67

+ 表示函数仅仅在 RTX51 Full 中具备。

RTX51 Full 里附加的调试和支持函数如表 3-6 所示。

表 3-6 RTX51 Full 附加函数

函 数	描 述
oi_reset_int_mask	禁止 RTX51 的外部中断资源
oi_set_int_mask	允许 RTX51 的外部中断资源
os_check_mailbox	返回指定信箱的状态信息
os_check_mailboxes	返回所有的系统信箱的状态信息
os_check_pool	返回内存池中的块信息
os_check_semaphore	返回指定信号量的状态信息
os_check_semaphores	返回所有的系统信号量信息
os_check_task	返回指定任务的状态信息
os_check_tasks	返回所有的系统任务的状态信息

CAN 函数仅在 RTX51 Full 中提供，CAN 控制器支持 Philip 82C200 和 80C592，以及 Intel 82526，如表 3-7 所示。

表 3-7 CAN 函数

CAN 函数	描 述
can_bind_obj	为一个任务绑定一个对象：当对象被接收时，任务启动
can_def_obj	定义通信对象
can_get_status	获取 CAN 控制器状态
can_hw_init	初始化 CAN 控制器硬件
can_read	直接读取一个对象的数据
can_receive	接收所有无界的对象
can_request	向一个指定的对象发送一个远程帧
can_send	通过 CAN 总线发送一个对象
can_start	开始 CAN 通信
can_stop	结束 CAN 通信
can_task_create	创建 CAN 通信任务
can_unbind_obj	断开任务和对象之间的绑定
can_wait	等待一个约束的对象被接收
can_write	向一个对象写入新数据，不用发送

下面列出了针对 RTX51 Tiny 的一些函数，并带有简要的说明和执行时间，如表 3-8 所示。

表 3-8 RTX51 Tiny 功能函数

函 数	描 述	执行周期数
Os_create_task	将某任务移入执行队列	302
Os_delete_task	从执行队列中移去某任务	172
Os_send_signal	发送一信号到某任务（从任务调用）	408（任务切换） 316（快速任务切换） 71（不含任务切换）
Os_clear_signal	删除一发送信号	57
Isr_send_signal	发送一信号到某任务（从中断调用）	46
os_running_task_id	返回当前执行的任务号	
os_wait	等待某事件	68（对未就绪信号） 160（对未就绪消息）
os_wait1	等待某事件	
os_wait2	等待某事件	

下面具体描述 RTX51 Tiny 系统函数。

3.3.1 信号控制函数

接下来首先介绍一下 RTX51 Tiny 中的信号控制函数。

（1）isr_send_signal

函数原型：

```
char isr_send_signal(unsigned char task_id);
```

功能说明：

发送一个信号到 `task_id` 说明的任务。如果此任务已在等待一个信号，那么调用函数将使此任务就绪，准备执行；否则，信号将存储在此任务的信号标志中。此函数只能从中断函数中调用。

返回值：

如果执行成功，此函数返回值为 0；如果所指定的任务不存在，则返回值为-1。

程序实例：

```
#include <rtx51tny.h>
void tst_isr_send_signal (void) interrupt 2
{
    isr_send_signal (6);           //信号任务 #6
}
```

(2) os_clear_signal

函数原型：

`char os_clear_signal(unsigned char task_id);`

功能说明：

清除由 `task_id` 说明的任务的信号。

返回值：

如果信号清除成功，此函数返回值为 0；如果所指定的任务不存在，则返回值为-1。

程序实例：

```
#include <rtx51tny.h>
#include <stdio.h>           //头文件
void tst_os_clear_signal (void) _task_ 8
{
    os_clear_signal (3);      //清除任务 5 的信号标志位
}
```

(3) os_send_signal

函数原型：

`char os_send_signal(unsigned char task_id);`

功能说明：

发送一个信号到 `task_id` 说明的任务。如果此任务已在等待一个信号，那么调用函数将使此任务就绪，准备执行；否则，信号将存储在此任务的信号标志中。此函数只能在任务函数中调用。

返回值：

如果执行成功，此函数返回值为 0；如果所指定的任务不存在，则返回值为-1。

程序实例：

```
#include <rtx51tny.h>
#include <stdio.h>           //头文件
void signal_func (void) _task_ 2
```

```

{
os_send_signal (6);                                //信号任务 #6
}
void tst_os_send_signal (void) _task_6
{
os_send_signal (2);                                //信号任务 #2
}

```

3.3.2 任务控制函数

下面介绍 RTX51 Tiny 中的任务控制函数。

(1) os_create_task

函数原型:

```
char os_create_task(unsigned char task_id);
```

功能说明:

启动已定义的由 task_id 说明的任务，此任务根据 RTX51 运行规则，标记为就绪，并准备执行。

返回值:

如果任务成功启动，此函数返回值为 0；如果没有 task_id 说明的任务，则返回值为-1。

程序实例:

```

#include <rtx51tiny.h>
#include <stdio.h>                                //头文件
void new_task (void) _task_2
{
}
void tst_os_create_task (void) _task_0
{
    if (os_create_task (2))
    {
        printf ("Couldn't start task 2\n");
    }
}

```

(2) os_delete_task

函数原型:

```
char os_delete_task(unsigned char task_id);
```

功能说明:

停止 task_id 说明的任务，此任务将从任务表中删除。

返回值:

如果任务成功启动，此函数返回值为 0；如果没有 task_id 说明的任务，则返回值为 -1。

程序实例：

```
#include <rtx51tiny.h>
#include <stdio.h>                                //头文件
void tst_os_delete_task (void) _task_ 0
{
    if (os_delete_task (2))
    {
        printf ("Couldn't stop task 2\n");
    }
}
```

(3) os_running_task_id

函数原型：

char os_running_task_id(unsigned char task_id);

功能说明：

判断当前执行任务的编号。

返回值：

返回当前正在执行的任务的编号，返回值为 0~15。

程序实例：

```
#include <rtx51tiny.h>
#include <stdio.h>                                //头文件
void tst_os_running_task (void) _task_ 3
{
    unsigned char tid;
    tid = os_running_task_id ();
    /* tid = 3 */
}
```

3.3.3 延时控制函数

下面介绍 RTX51 Tiny 中的延时控制函数。

(1) os_wait

函数原型：

```
char os_wait(unsigned char event_sel,           //将要等待的事件
             unsigned char ticks,               //将要等待的定时器时标数
             unsigned char dummy);              //未用参数
```

功能说明：

停止当前执行的任务，并等待一个或多个事件，如时间间隔、超时、从另一个任务或中断发出的信号等，参数 event_sel 说明所等待的一个事件或几个事件的组合，事件的种类如表 3-9 所示。

表 3-9 os_wait 函数等待事件常数表

事 件 常 数	文 字 说 明
K_IVL	等待一个报时信号间隔
K_SIG	等待一个信号
K_TMO	等待一个超时(time-out)

上述事件可以用字符 “|” 进行逻辑或。例如 K_TMO | K_SIG 规定任务等待一个超时或一个信号。参数 `tasks` 规定等待一个间隔事件 (`k_ivl`) 或一个超时事件 (`k_tmo`) 的报时信号数目。参数 `dummy` 是为了提供与 RTX51 的兼容性，在 RTX51 Tiny 中没有使用。

返回值：

当一个指定的事件发生时，任务允许运行，运行被恢复，并且 `os_wait` 函数返回一个用于识别重新启动任务的事件的识别常数，可能的返回值是如表 3-10 所示。

表 3-10 os_wait 函数返回值

返 回 值	文 字 说 明
SIG_EVENT	接收到一个信号
TMO_EVENT	一个超时 (time-out) 已经完成或一个间隔 (interval) 已经期满
NOT_OK	参数 <code>event_sel</code> 的值无效

程序实例：

```
#include <rtx51tny.h>
#include <stdio.h>                                     //头文件
void tst_os_wait (void) _task_ 9
{
    while (1)
    {
        char event;
        event = os_wait (K_SIG + K_TMO, 50, 0);
        switch (event)
        {
            default:
                break;
            case TMO_EVENT:                               //超时事件
                //50 倍间隔的超时发生
                break;
            case SIG_EVENT:                               //信号事件
                //信号接收
                break;
        }
    }
}
```

(2) `os_wait1`

函数原型:

```
char os_wait1(unsigned char event_sel);
```

功能说明:

暂停当前任务，等待一个事件的发生。它是 `os_wait` 函数的一个子集，不接受针对 `os_wait` 函数提供的全部事件。参数 `event_sel` 指定了等待的事件，它只能是 `K_SIG`。

返回值:

当信号事件发生时，任务就被允许执行，任务的执行将恢复。返回用于识别事件、使任务重新启动的常量，可能的返回值如表 3-11 所示。

表 3-11 `os_wait1` 函数返回值列表

返 回 值	文 字 说 明
<code>SIG_EVENT</code>	接收到一个信号
<code>NOT_OK</code>	参数 <code>event_sel</code> 的值无效

(3) `os_wait2`

函数原型:

```
char os_wait2(unsigned char event_sel,          //将要等待的事件
              unsigned char tasks);             //将要等待的定时器时标数
```

功能说明:

与 `os_wait` 相同，但是不需要 `dummy` 参数。

返回值:

与 `os_wait` 相同。

3.4 建立RTX51 Tiny应用程序

编写 RTX51 Tiny 程序要求把 `rtx51tny.h` 头文件包含在 C 语言程序的 `\c51\inc\` 子目录下，而且使用 `_task_` 函数属性声明任务。RTX51 Tiny 程序不需要一个 C 语言主函数 `main()` 下，链接过程将包含首先执行任务 0 的程序代码。

1. 编译 RTX51 Tiny 程序

RTX51 Tiny 应用程序不需要专门的编译程序，可以像编译普通的 C 语言源文件一样编译 RTX51 Tiny 源文件。

2. 链接RTX51 Tiny 程序

RTX51 Tiny 应用程序必须使用 BL51 code banking linker/locator 进行链接，必须在所有目标文件后，在命令行上规定 RTX51 Tiny 指令。

3. 优化RTX51 Tiny程序

在建立 rtx51 应用程序时，应该注意以下事项：

① 对于多重任务应尽可能采用 `os_wait` 函数触发，而不用时间片轮。因为使用时间片轮切换任务需要 13 B 的堆栈空间来存储任务环境（寄存器等），如果由 `os_wait` 函数触发，则不需要环境存储器。`os_wait` 函数还会改善系统反应时间。

② 不要将报时信号的中断速率设得太快。因为，每个时钟报时中断需要 100~200 个 CPU 周期，所以应该把时间报时信号速率设得足够高，以使中断等待时间减到最少。

第 4 章 常用的单片机芯片介绍

本章主要介绍几个常用的 8 位单片机系列芯片。通过对这些芯片的了解，读者可以在实际应用中更加灵活地选择 CPU，从而在开发过程中节省实际的开销。

4.1 HOLTEK公司HT48XX系列单片机介绍

HT48 系列单片机是 8 位高性能的精简指令单片机（RISC），专门为多输入/输出的产品而设计，尤其适用于遥控器、电扇/灯光控制器、洗衣机控制器、电子秤、玩具及各种子系统的控制器，它们都有一个暂停特性来降低功耗。这些单片机的主要不同之处在于它们各自所具有的 ROM 和 RAM 的大小不同、定时/计数器的个数和长度不同、输入/输出的数目不同和堆栈级数的不同。

C 型单片机和 R 型单片机的区别在于 C 型是掩模型，R 型是一次性烧录型，表 4-1 为 HT48 系列单片机各型号之间内部资源的比较。

表 4-1 HT48 系列单片机各型号之间内部资源的比较

产 品 型 号	程序存储器 ROM	RAM	定 时 器	I/O 线	最快指令周期 /μs	内部 中断	外部 中断	堆栈
HT48C10-1/HT48R10A-1	1 KB×14	64×8	1 (8)	21	0.5	1	1	4
HT48C30-1/HT48R30A-1	2 KB×14	96×8	1 (8)	25	0.5	1	1	4
HT48C50-1/HT48R50A-1	4 KB×15	160×8	2 (8,16)	35	0.5	2	1	6
HT48C70-1/HT48R70A-1	8 KB×16	224×8	2(16,16)	56	0.5	2	1	16
HT48C05/HT48R05A-1	0.5 KB×14	32×8	1 (8)	13	0.5	1	1	2
HT48C06/HT48R06A-1	1 KB×14	64×8	1 (8)	13	0.5	1	1	2
HT48CA0-1/HT48RA0-1	1 KB×14	32×8	无	17	1	0	0	1
HT48CA0-2/HT48RA0-2	1 KB×14	32×8	无	15	1	0	0	1
HT48CA1/HT48RA1	8 KB×14	224×8	2 (8,16)	23	0.5	2	1	8
HT48CA3/HT48RA3	24 KB×16	224×8	2 (8,16)	23	0.5	2	1	8
HT48CA5/HT48RA5	40 KB×16	224×8	2 (8,16)	23	0.5	2	1	8

4.1.1 HT48R05A-1

HT48R05A-1 是基本型高性能 8 位单片机，它有多种封装形式，一种是 16 条引脚的 SSOP 封装，另外一种 DIP 封装形式，还有一种是 18 条引脚 SOP 封装。它的 18 脚 SOP 封装的引脚如图 4-1 所示，其内部结构如图 4-2 所示。

当时钟频率为 8 MHz 时，指令周期仅为 0.5 μs，一个 8 位带溢出中断功能 8 级预分频的定时/计数器，特别适合于多端口的控制系统、如遥控器、家用电器及其他电子系统，其主

要功能特性如表 4-2 所示。

表 4-2 HT48R50A-1 的特征

工作电压：2.2~5.5V	暂停及端口唤醒功能
13 条双向 I/O 输入/输出口线	低电压复位功能
一个 8 位可编程定时/计数器	内置石英晶体和 RC 振荡器
63 条精简指令集	看门狗（WDT）电路
32×8 位内部数据存储器 RAM	支持 PFD 蜂鸣器输出端
512×14 位程序存储器 ROM	所有指令均为 1~2 个机器周期
14 位查表指令	2 级堆栈
位操作指令	中断输入与一个 I/O 口复用

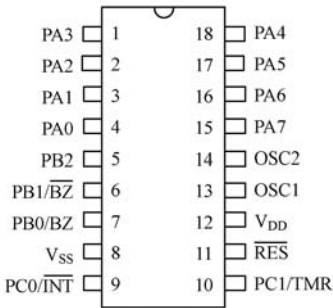


图 4-1 HT48R05A-1 的引脚图

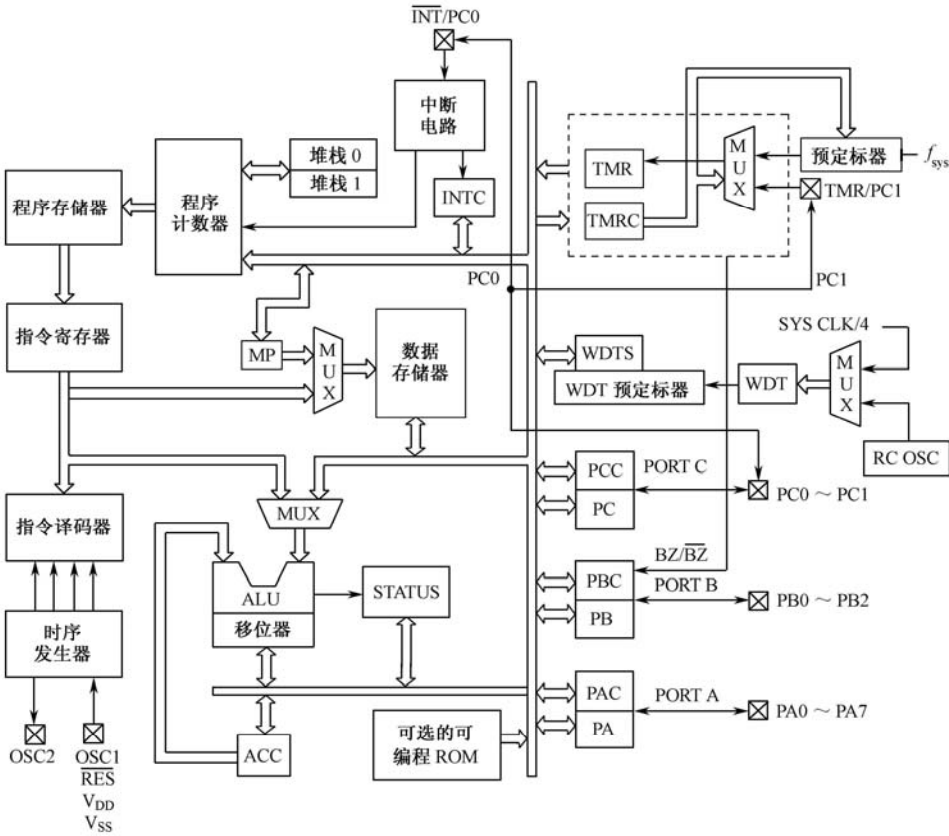


图 4-2 HT48R05A-1 的内部结构框图

4.1.2 HT48R50A-1

HT48R50A-1 是多 I/O 口高性能 8 位单片机，它有两种封装形式，一种是 28 条引脚的封装（28kSKDIP/SOP），另外一种 48SSOP 封装形式，除了双向 I/O 口线的数量不同外，其他功能都基本一致，其 28 个引脚的封装如图 4-3 所示，其内部结构如图 4-4 所示，其主

要的功能特性如表 4-3 所示。

表 4-3 HT48R50A-1 主要功能特性

工作电压：3.3~5.5 V	暂停及端口唤醒功能
23/35 条双向 I/O 输入/输出口线	低电压复位功能
一个 16 位可编程定时/计数器	一个内置晶体和 RC 振荡器
一个 8 位带溢出中断 8 级预分频定时/计数器	看门狗（WDT）电路
160×8 位内部数据存储器 RAM	一个蜂鸣器及 PFD 输出
4 K×14 位程序存储器 ROM	内置 32.768 kHz 的 RC 振荡器
15 位查表指令	6 级堆栈
63 条精简指令集	所有指令均为 1~2 个指令周期

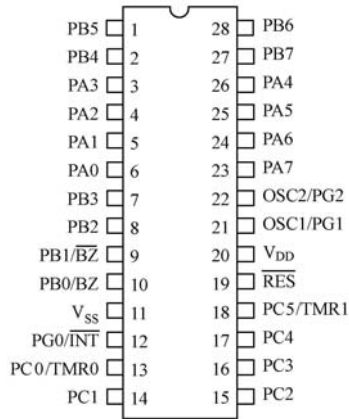


图 4-3 HT48R50A-1 引脚图

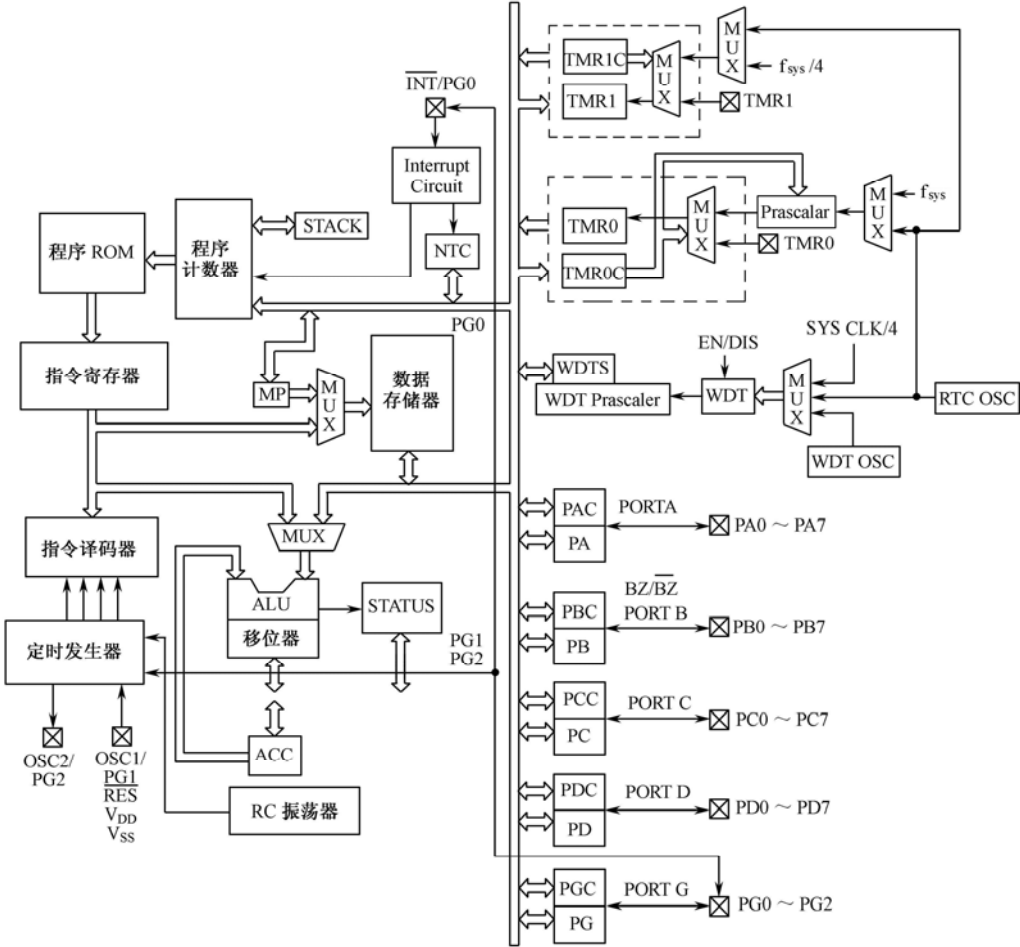


图 4-4 HT48R50A-1 内部结构框图

4.1.3 HT48C50-1

HT48C50-1 和 HT48R50A-1 基本一样，都是多 I/O 口高性能 8 位单片机，它有两种封装形式，一种是 28 条引脚的封装（28kSKDIP/SOP），另外一种 48SSOP 封装形式，除了双向 I/O 口线的数量不同外，其他功能都基本一致，其 28 条引脚的封装如图 4-5 所示，其内部结构如图 4-6 所示，其主要的功能特性如表 4-4 所示。

表 4-4 HT48C50-1 主要的功能特性

工作电压：3.3~5.5 V	暂停及端口唤醒功能
23/35 条双向 I/O 输入/输出口线	低电压复位功能
一个 16 位可编程定时/计数器	一个内置晶体和 RC 振荡器
一个 8 位带溢出中断 8 级预分频定时/计数器	看门狗（WDT）电路
160×8 位内部数据存储器 RAM	一个蜂鸣器及 PFD 输出
4 K×14 位程序存储器 ROM	内置 32.768 kHz 的 RC 振荡器
15 位查表指令	6 级堆栈
63 条精简指令集	所有指令均为 1~2 个指令周期

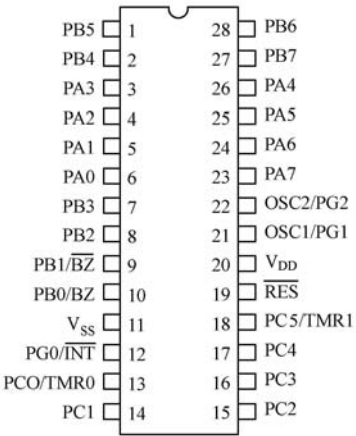


图 4-5 HT48C50-1 的引脚图

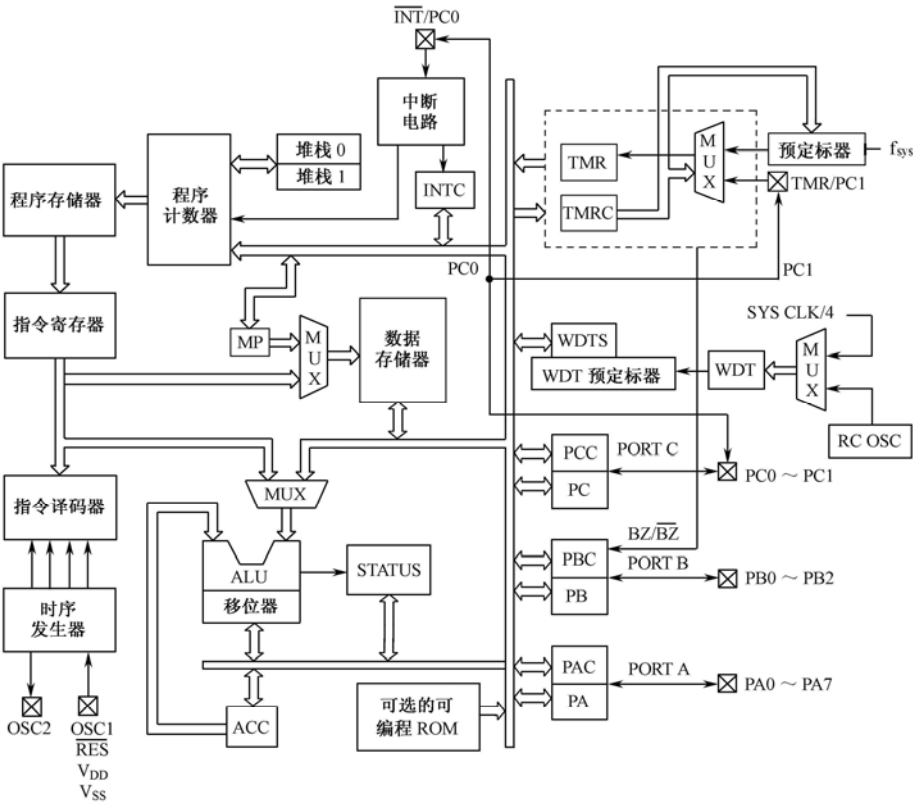


图 4-6 HT48C50-1 的内部结构框图

HT48C50-1 和 HT48R50A-1 唯一区别在于前者是掩模型的，而后者是烧录型的。

上述三个芯片是该系列比较有代表性的，其余的芯片请读者参阅相关的技术资料自行查阅。

4.2 Motorola公司的MC68HC08 系列单片机

Motorola 公司 1999 年推出了 8 位微控制器 MC68HC08 系列单片机，所有 MC68HC08 系列单片机都使用增强型 MC68HC08 的中央处理单元（CPU08），具有速度快、功能强、功耗小及价格低等特点。

4.2.1 MC68HC08AS32CFN

MC68HC08AS32CFN 是 MC68HC08 系列中的一款高性价比的 8 位单片机，其特性如表 4-5 所示，其 64 位引脚和内部结构分别如图 4-7 和图 4-8 所示。

表 4-5 MC68HC08AS32CFN 的特性

高效的 MC68HC08 架构	完全兼容 M146805 的 MC68HC05 系列
8.4 MHz 的内部总线频率	32 265 位的 ROM
512 位的片上 E ² PROM	1 MB 的片上 RAM
内置串行外设接口模块（SPIM）	内置通信接口模块（SCIM）
16 位的 6 通道定时接口模块（TIM）	内置时钟发生模块（CGM）
8 位 15 通道的 A/D 转换器（ADC）	主复位引脚以及上电复位

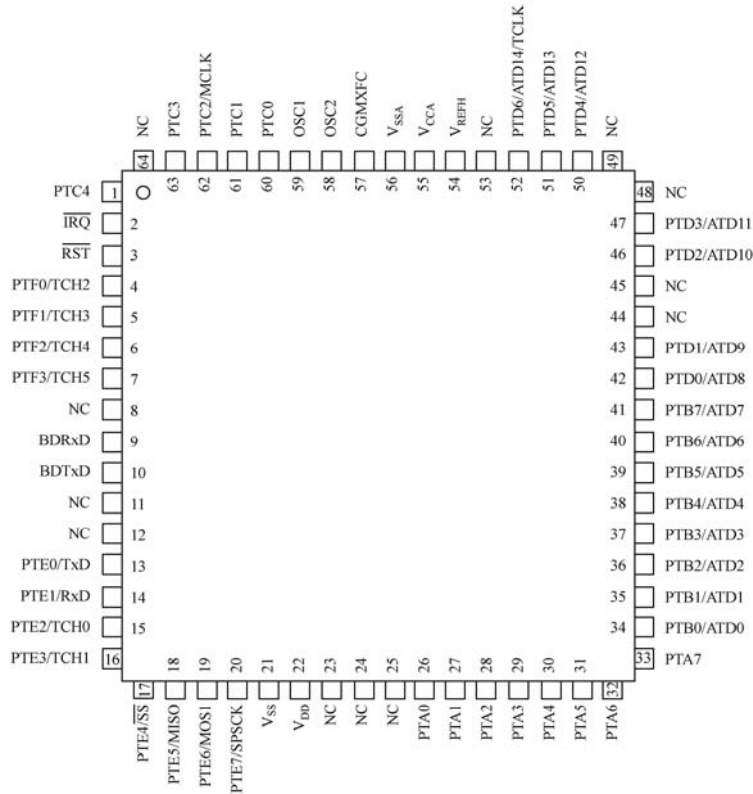


图 4-7 MC68HC08AS32CFN 引脚图

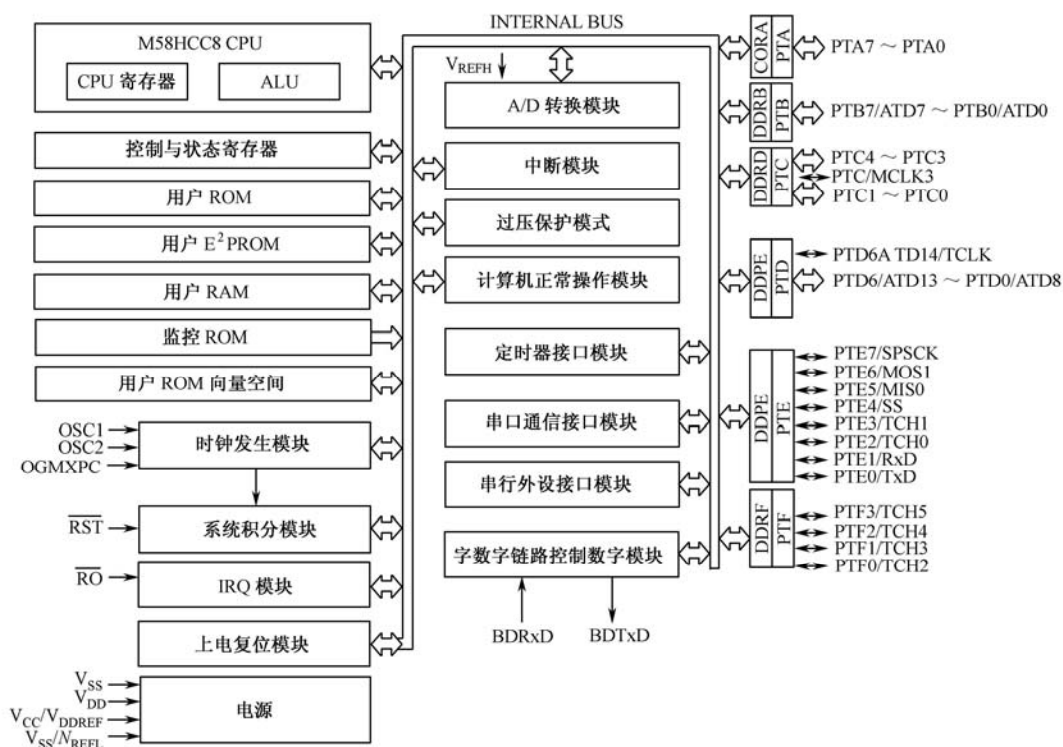


图 4-8 MC68HC08AS32CFN 的内部结构框图

4.2.2 MC68HC08AS32FU

MC68HC08AS32FU 的特性如表 4-6 所示，其 52 位引脚和内部结构分别如图 4-9 和图 4-10 所示。

表 4-6 MC68HC08AS32FU 的特性

低功耗设计（停止和待机两种状态）	内置时钟发生模块（CGM）
8.4 MHz 的内部总线频率	主复位引脚以及上电复位
512 位的片上 E²PROM	32 265 位的 ROM
内置串行外设接口模块（SPIM）	1 MB 的片上 RAM
16 位的 6 通道定时接口模块（TIM）	内置通信接口模块（SCIM）

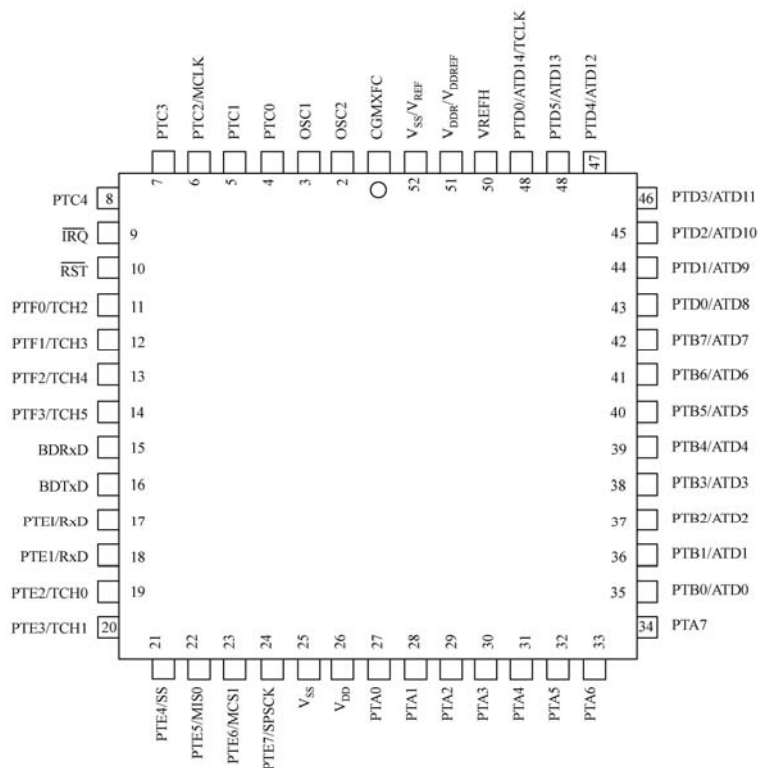


图 4-9 MC68HC08AS32FU 引脚图

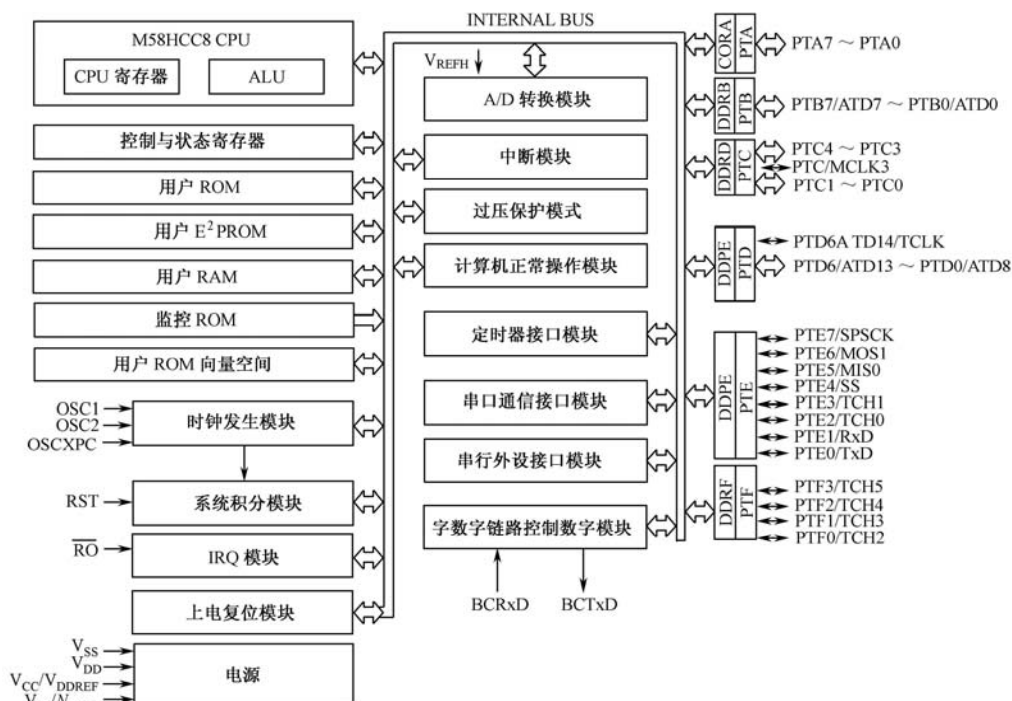


图 4-10 MC68HC08AS32FU 的内部结构框图

上述两种芯片是 Motorola 公司 MC68HC08 系列单片机中比较有代表性的，同系列的其他芯片的参考资料可查询 Motorola 公司的技术文档。

4.3 Philips公司推出的改进型C51 单片机

该系列单片机有 P89C51X2 系列、P89C52X2 系列、P89C 54X2 系列、P89C 58X2 系列，它们比传统的 C51 单片机在性能上有了很多改进，如时钟支持 12 时钟和 6 时钟操作，其工作频率范围：6 时钟模式时为 0~20 MHz；12 时钟模式时为 0~33 MHz，更突出的是由 Flash（程序存储器的内容至少可以改写 1 000 次）存储器取代了原来的 ROM（一次性写入）。

4.3.1 产品性能

其具体的特性如下：

- 最高工作频率为 33 MHz，大家都知道 89C51 的极限工作频率是 24 MHz，就是说 S51 具有更高工作频率，从而具有了更快的计算速度，存储器寻址范围：64 KB ROM 和 64 KB RAM；
- 电源控制模式：时钟可停止和恢复、空闲模式、掉电模式；
- LQFP、PLCC 或 DIP 封装；
- 3 个加密位，这使得对 89S51 的解密变为不可能，程序的保密性得到极大加强，这样就可以有效地保护知识产权不被侵犯；
- 4 个中断优先级，6 个中断源，4 个 8 位 I/O 口；
- 内部集成看门狗计时器，不再需要像 89C51 那样外接看门狗计时器单元电路；
- 3 个 16 位定时/计数器 T0、T1 标准 80C51 和增加的 T2 捕获和比较；
- 可编程时钟输出；
- 异步端口复位；
- 低 EMI（禁止 ALE 以及 6 时钟模式）；
- 掉电模式可通过外部中断唤醒。

表 4-7 为操作模式、电源电压以及最大外部时钟频率之间的关系。

表 4-7 操作模式、电源电压以及最大外部时钟频率之间的关系

操 作 模 式	电 源 电 压	最大外部时钟频率/MHz
6 时钟模式	5 V±10%	22
12 时钟模式	5 V±10%	33

4.3.2 内部框图及引脚说明

图 4-11 和图 4-12 分别是该系列 CPU 的内部结构框图和单片机内部模块框图。

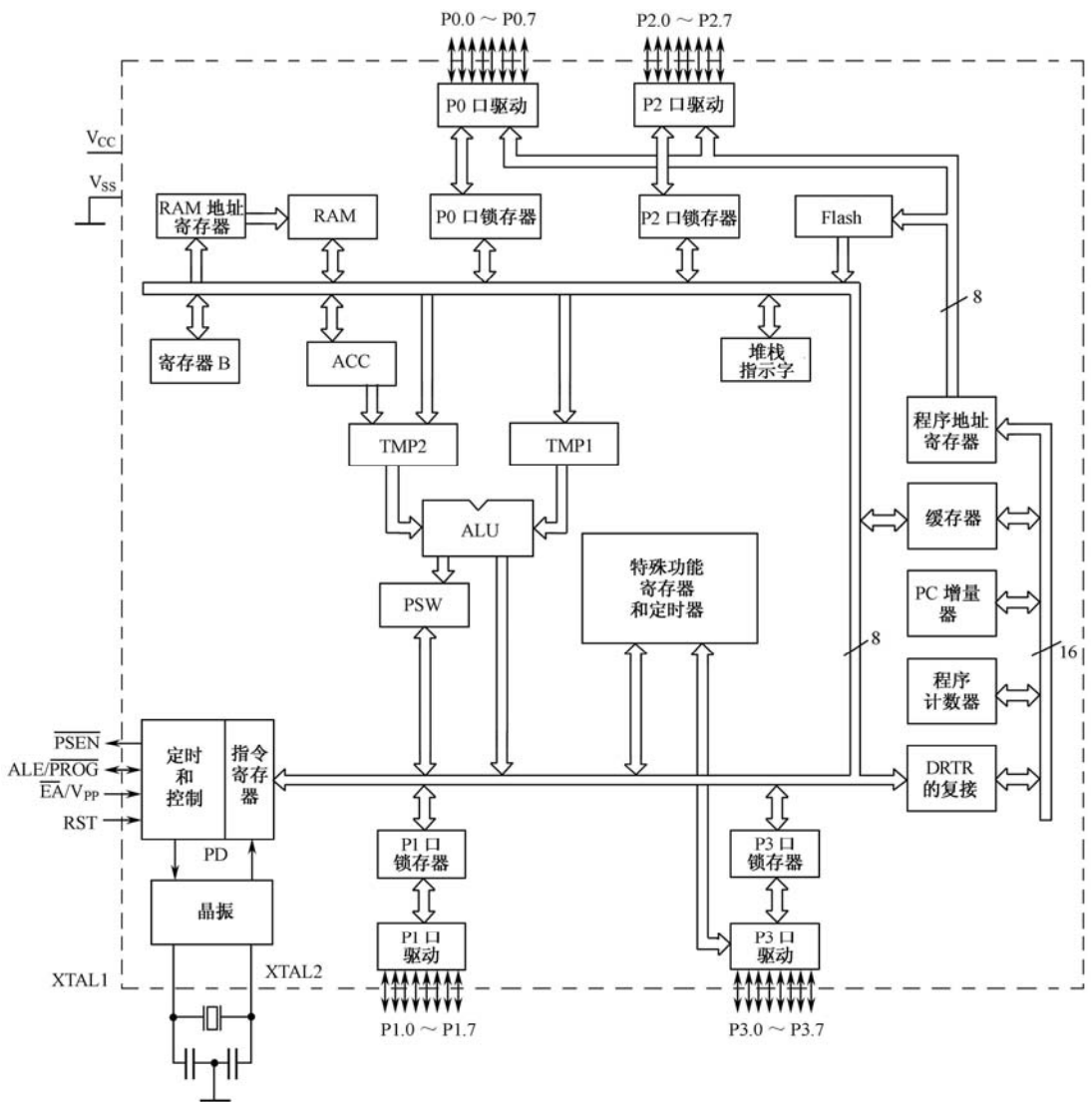


图 4-11 CPU 内部框图

该系列单片机有三种封装形式：DIP、LCC、QFP。本书主要介绍应用最为广泛的 DIP 型封装，其他两种封装形式读者可以参考 Philips 公司的技术资料，图 4-13 为 DIP 封装引脚图。

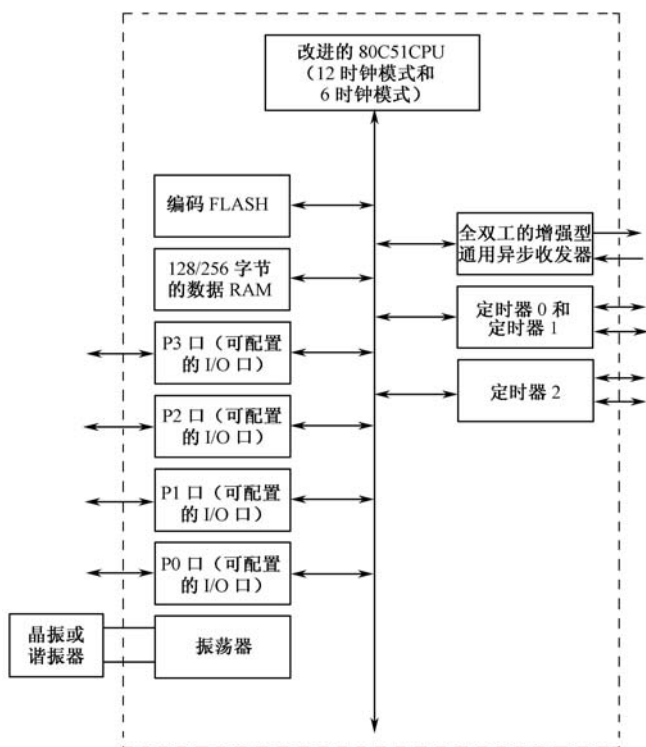


图 4-12 内部模块框图

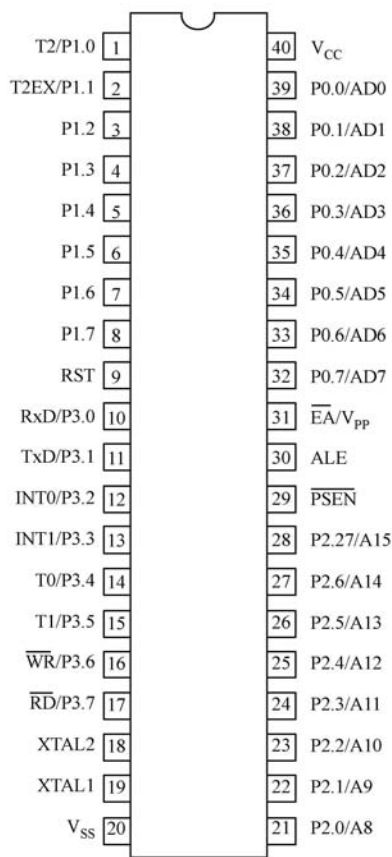


图 4-13 改进型 C51 单片机系列
DIP 封装引脚图

该系列引脚描述如表 4-8 所示。

表 4-8 改进型 C51 单片机系列引脚描述

引脚号	引脚名称	功能
20	V _{SS}	地
40	V _{CC}	电源提供掉电空闲正常工作电压
39~32	P0.0~P0.7	P0 口是开漏双向口，可以写为 1 使其状态为悬空，用作高阻输入，P0 口也可以在访问外部程序存储器时作为地址的低字节，在访问外部数据存储器时作为数据总线，此时通过内部强上拉输出 1
1~8	P1.0~P1.7	P1 口是带内部上拉电阻的双向 I/O 口，向 P1 口写入 1 时，P1 口被内部上拉为高电平，可用做输入口。当做为输入脚时，被外部拉低的 P1 口会因为内部上拉电阻作用而输出电流（见 DC 电气特性），P1 口第 2 功能：T2（P1.0）定时/计数器 2 的外部计数输入/时钟输出（见可编程输出）；T2EX（P1.1）定时/计数器 2 重载/捕捉/方向控制
21~28	P2.0~P2.7	P2 口是带内部上拉电阻的双向 I/O 口，向 P2 口写入 1 时，P2 口被内部上拉为高电平，可用做输入口。当做为输入脚时，被外部拉低的 P2 口会因为内部上拉电阻作用而输出电流（见 DC 电气特性）。在访问外部程序存储器和外部数据时分别作为地址高位字节和 16 位地址（MOVX @DPTR）此时通过内部强行上拉传送 1。当使用 8 位寻址方式（MOV@Ri）访问外部数据存储器时，P2 口发送 P2 特殊功能寄存器的内容

引脚号	引脚名称	功能
10~17	P3.0~P3.7	P3 口是带内部上拉电阻的双向 I/O 口，向 P3 口写入 1 时，P3 口被内部上拉为高电平，可用做输入口。当做为输入脚时，被外部拉低的 P3 口会因为内部上拉电阻作用而输出电流（见 DC 电气特性）
		P3 口还具有以下特殊功能：
10		RXD（P3.0）：串行输入口；
11		TXD（P3.1）：串行输出口；
12		INT0（P3.2）：外部中断 0；
13		INT1（P3.3）：外部中断；
14		T0（P3.4）：定时器 0 外部输入；
15		T1（P3.5）：定时器 1 外部输入；
16		WR（P3.6）：外部数据存储器写信号；
17		RD（P3.7）：外部数据存储器读信号
9	RST	复位：当晶振在运行中，只要复位引脚出现 2 个机器周期高电平即可复位。内部有扩散电阻连接到 V _{SS} ，仅需要外接一个电容到 V _{CC} 即可实现上电复位
30	ALE	地址锁存使能：在访问外部存储器时，输出脉冲锁存地址的低字节；在正常情况下，ALE 输出信号恒定为 1/6 振荡频率，并可用做外部时钟或定时，注意每次访问外部数据时，一个 ALE 脉冲将被忽略。ALE 可以通过置位 SFR 的 auxiliary.0，禁止置位后 ALE 只能在执行 MOVX 指令时被激活
29	$\overline{\text{PSEN}}$	程序存储使能：当执行外部程序存储器代码时， $\overline{\text{PSEN}}$ 每个机器周期被激活两次；在访问外部数据存储器时， $\overline{\text{PSEN}}$ 访问内部程序存储器时 $\overline{\text{PSEN}}$ 无效
31	$\overline{\text{EA}}/\text{VPP}$	外部寻址使能/编程电压在访问整个外部程序存储器时 $\overline{\text{EA}}$ 必须外部置低。如果 $\overline{\text{EA}}$ 为高电平时将执行内部程序，除非程序计数器包含大于片内 Flash 的地址该引脚在对 Flash 编程时接 5V/12V 编程电压（V _{pp} ）；如果保密位为 1，已编程 $\overline{\text{EA}}$ 在复位时由内部锁存
19	XTAL1	晶体 1：反相振荡放大器输入和内部时钟发生电路输入
18	XTAL2	晶体 2：反相振荡放大器输出

该系列单片机的其他性能指标等参数请参考 Philips 公司的技术资料。

4.4 Atmel 公司的 AT89S 系列单片机

Atmel 公司推出的可在系统编程的 C51 兼容单片机 AT89S51/52 将全面替代 AT89C51/52 单片机。

4.4.1 AT89S 系列单片机的特点

AT89S 系列单片机有以下特点：

- 兼容 C51 微控制器；
- 4/8 KB Flash 存储器支持在系统编程 ISP1000 次擦写周期；
- 128/256 B 片内 RAM；
- 工作电压 4.0~5.5 V；
- 全静态时钟 0~33 MHz；
- 3 级程序加密，使得解密的可能性急剧降低，很好地保护了知识产权；

- 32 个可编程 I/O 口；
- 2/3 个 16 位定时/计数器；
- 6/8 个中断源；
- 全双工通用异步收发器（UART）；
- 完全的双工 UART 串行口；
- 低功耗支持空闲模式（Idle）和掉电模式（Power-down）；
- Power-down 模式支持中断唤醒；
- 内置看门狗定时器；
- 上电复位标志。

AT89S51/52 和 AT89C 相比，新增加了以下功能：

- 支持在系统编程 ISP 生产及维护更方便；
- 增加了片内看门狗电路，使用户的应用系统更坚固；
- 双数据指针使数据操作更加快捷方便；
- 速度更高最高可使用 33 MHz 的晶振；
- 尽管 AT89S 系列单片机新增加了不少功能，但用户也可以直接替换应用系统中的 AT89C51/52 软件硬件，而不需进行任何修改；
- 三种封装形式：DIP、TQFP 和 PLCC。

4.4.2 AT89S系列单片机的引脚图及内部结构框图

以 AT89S51 为例介绍该系列单片机，其他型号可参考 Atmel 公司的技术资料。AT89S51 有三种封装形式：DIP、TQFP 和 PLCC，其 DIP 型封装的引脚如图 4-14 所示，AT89S51 的内部结构如图 4-15 所示。

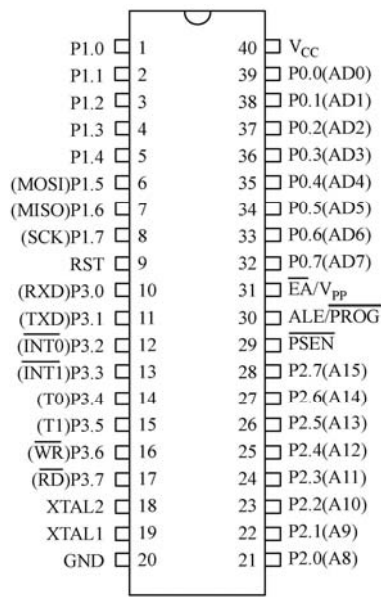


图 4-14 AT89S51 DIP 型封装引脚图

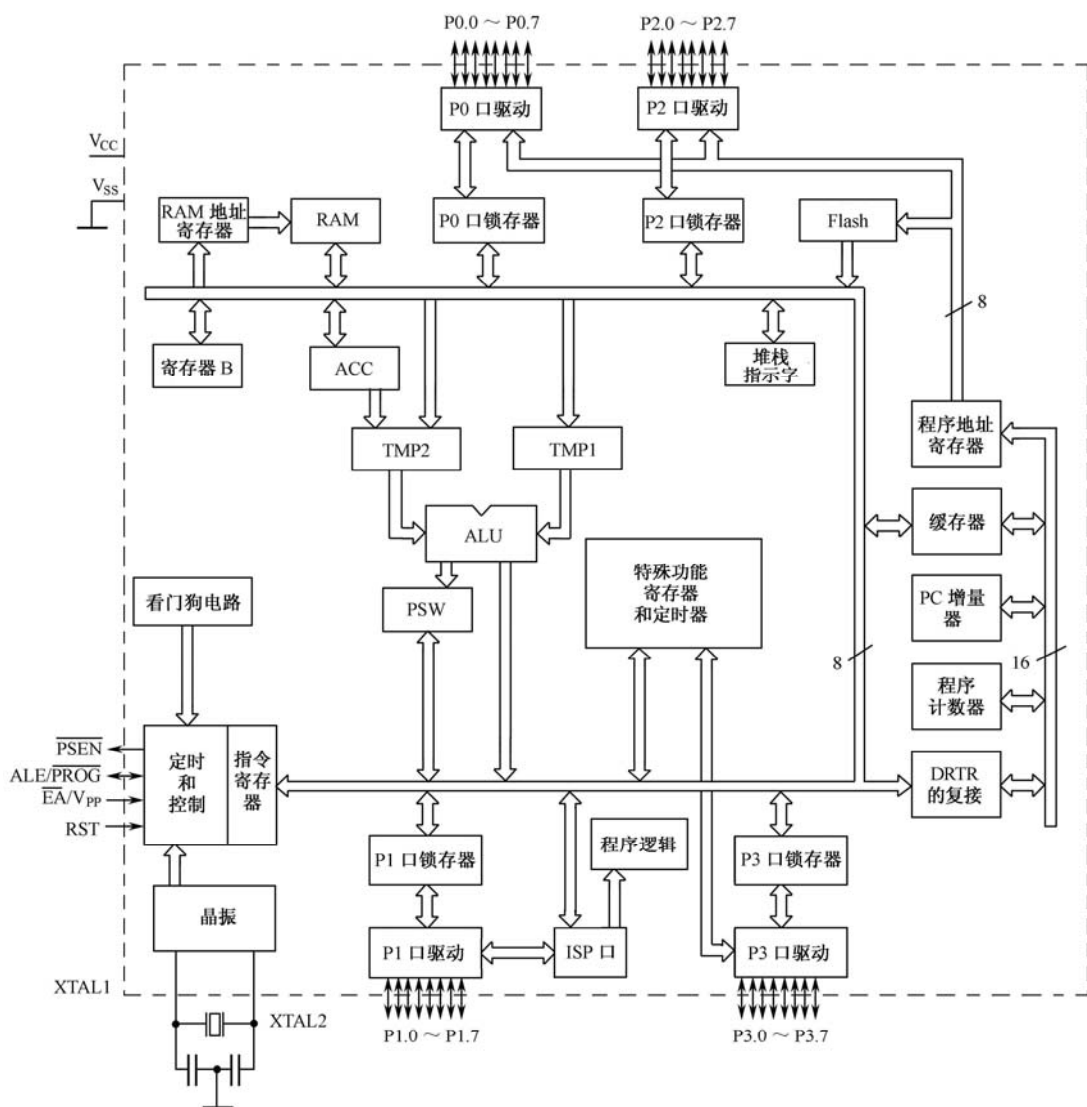


图 4-15 AT89S51 内部结构框图

第 5 章 键盘与显示实例

键盘和显示系统是单片机应用中人机交互的部分，具有很重要的地位，也是最常见的设计工作，本章将介绍几种典型的设计实例。

5.1 七段数码管显示

本实例将主要介绍单片机系统中极为简单，但又极为常用的数码管显示。新型数码管具有功耗低、亮度高、寿命长、尺寸小等优点，在系统中的主要作用是显示单片机的输出状态、数据等，因此在家电及工业控制中有着广泛的应用，经常用来显示温度、数字、重量、日期、时间等，最常见的有银行的利率显示屏、新颖的电子万年历、各种安全提示牌等。

5.1.1 实例说明

本例将利用 C51 单片机通过几款必要的器件，实现 4 位七段共阴极数码管动态显示 1、2、3、4。

5.1.2 七段数码管介绍

最常见的半导体数码管是由 7 个条状的发光二极管（LED）排列而成的，可实现数字“0~9”及少量字符的显示。为了显示小数点，另外增加了 1 个点状的发光二极管，因此数码管就由 8 个 LED 组成，我们分别把这些发光二极管命名为 a、b、c、d、e、f、g、dp，每一个 LED 称为一个字段，dp 为小数点，排列顺序以及引脚分布如图 5-1 所示。

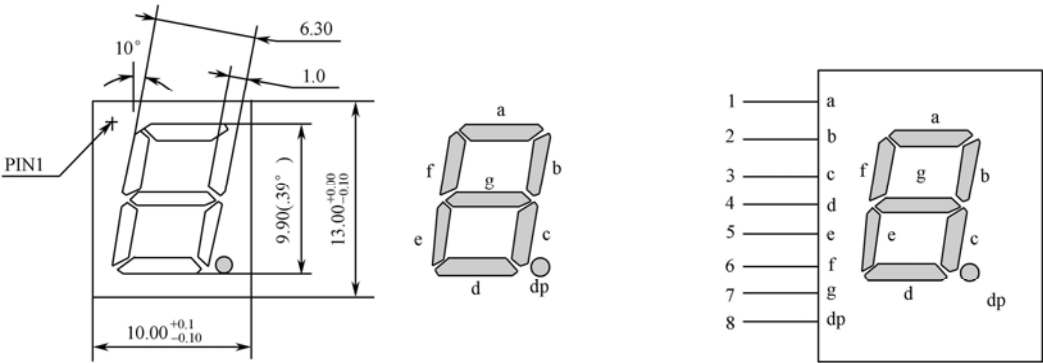


图 5-1 数码管的排列及引脚图

按照数码管上各发光二极管的电极连接方式的不同，可以将数码管分为共阳极数码管和共阴极数码管两种。

共阴极数码管是指把 a、b、c、d、e、f、g 这 7 个发光二极管的阴极连接到一起，形成

公共阴极（COM）的数码管。共阴极数码管在应用时应将公共极 COM 接到地线 GND 上，当某一字段发光二极管的阳极为高电平时，相应字段就发光；当某一字段的阳极为低电平时，相应字段就不亮。共阴极数码管内部连接如图 5-2 所示。

共阳极数码管是指把 a、b、c、d、e、f、g 这 7 个发光二极管的阳极连接到一起，成公共阳极（COM）的数码管。共阳极数码管在应用时应将公共极 COM 接到 +5V 电源上，当某一字段发光二极管的阴极为低电平时，相应字段就发光；当某一字段的阴极为高电平时，相应字段就不亮。共阳极数码管内部连接如图 5-3 所示。

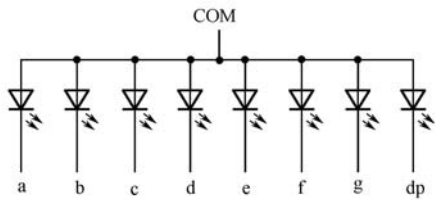


图 5-2 共阳极数码管内部连接图

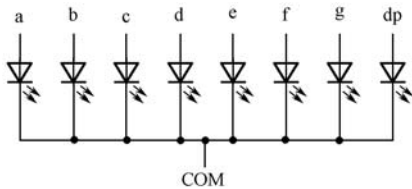


图 5-3 共阴极数码管内部连接图

数码管要正常显示，就要用驱动电路来驱动数码管的各个段码，用单片机驱动 LED 数码管有很多方法，按显示方式可分为静态显示和动态（扫描）显示，按译码方式可分为硬件译码和软件译码。

静态显示驱动也称直流驱动，是指每个数码管的每一个字段都由一个单片机的 I/O 口进行驱动，或者使用 BCD 码二-十进制译码器译码进行驱动，该接口用于笔画段字形代码。这样单片机只要把显示的字型代码发送到接口电路，该字段就可以显示发送的字型。静态显示电路具有输出锁存功能，单片机将显示的数据送出后就不再控制 LED，直到下一次显示时再传送一次新的显示数据。静态驱动的优点是编程简单，显示亮度高，显示数据稳定，占用 CPU 时间很少；缺点是占用 I/O 口多，如驱动 5 个数码管静态显示则需要 40 根 I/O 端口来驱动。

动态显示驱动是将所有数码管的 8 个显示笔画 a、b、c、d、e、f、g、dp 的同名端连在一起，另外为每个数码管的公共极 COM 增加位选通控制电路，位选通由各自独立的 I/O 线控制。当单片机输出字型码时，所有数码管都接收到相同的字型码，但究竟是哪个数码管会显示出字形，取决于单片机对位选通 COM 端电路的控制，所以只需将要显示的数码管的选通控制打开，该位就显示出字形，没有选通的数码管就不会发光。通过分时轮流控制各个数码管的 COM 端，就使各个数码管轮流受控显示，这就是动态显示驱动。动态显示占用 CPU 时间多，显示数据有闪烁感，但能够节省大量的 I/O 口，硬件开销小，而且功耗更低，可以降低成本和电源功耗。

硬件译码就是显示的段码完全由硬件完成，CPU 只要送出标准的 BCD 码即可，硬件接线有一定标准。软件译码是用软件来完成硬件的功能，硬件简单，接线灵活，显示字段完全由软件来处理，是目前常用的显示驱动方式。

5.1.3 硬件电路设计

单片机七段数码管动态显示电路的原理如图 5-4 所示，其中 P0 口是字段码，低电平有

效，P2 口是位码，高电平有效，P2.0 口控制第 1 个数码管，依次类推，P2.7 口控制第 8 个数码管。

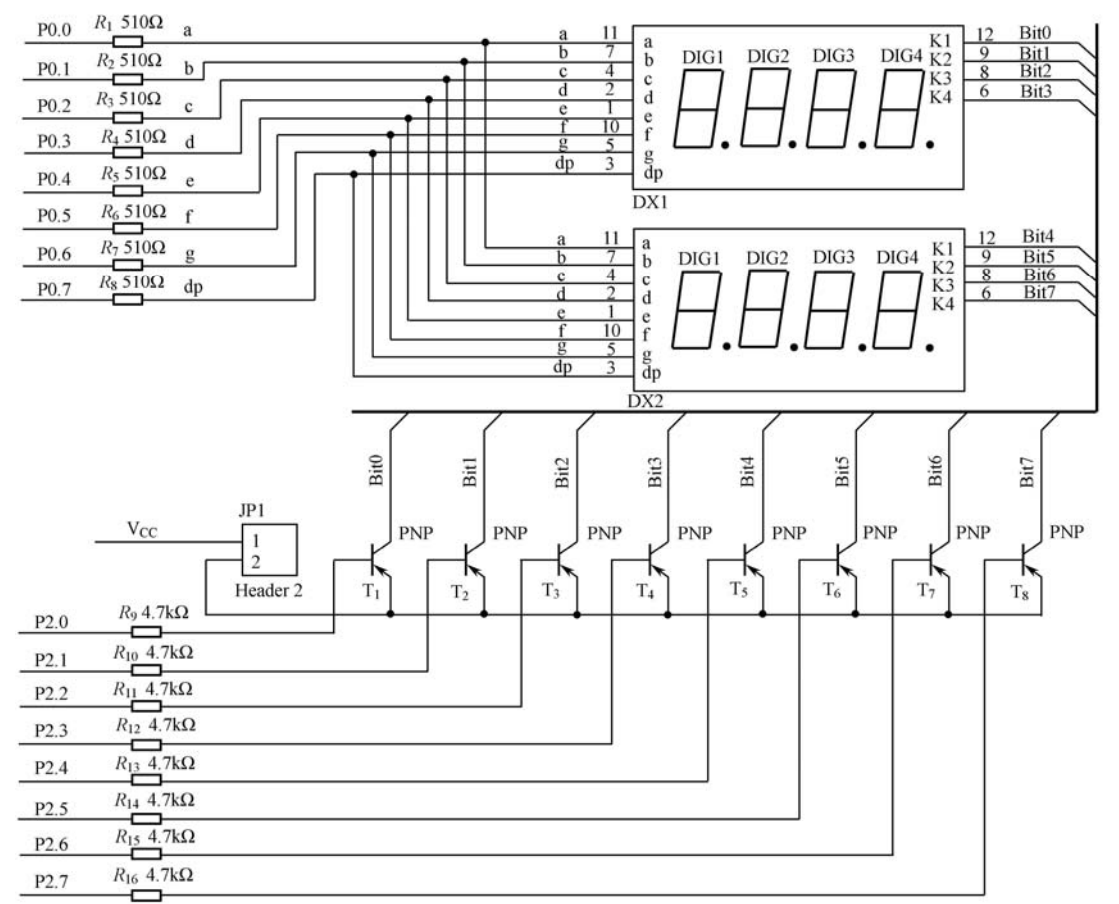


图 5-4 单片机七段数码管动态显示电路的原理图

七段数码管可以显示包括小数点在内的 0~9 个数字和部分英文字母，数码管各字段所加的电压不同，编码页不一样，获得的字型也不同，显示的字型、字段和编码的关系如表 5-1 所示。

表 5-1 字型、字段和编码的关系

字 型	dp g f e d c b a	编 码 (共阴极)
	D ₇ D ₆ D ₅ D ₄ D ₃ D ₂ D ₁ D ₀	
0	0 0 1 1 1 1 1 1	3FH
1	0 0 0 0 0 1 1 0	06H
2	0 1 0 1 1 0 1 1	5BH
3	0 1 0 0 1 1 1 1	4FH
4	0 1 1 0 0 1 1 0	66H
5	0 1 1 0 1 1 0 1	6DH
6	0 1 1 1 1 1 0 1	7DH

续表

字 型	dp g f e d c b a	编 码 (共阴极)
	D ₇ D ₆ D ₅ D ₄ D ₃ D ₂ D ₁ D ₀	
7	0 0 0 0 0 1 1 1	07H
8	0 1 1 1 1 1 1 1	7FH
9	0 1 1 0 1 1 1 1	6FH
A	0 1 1 1 0 1 1 1	77H
B	0 1 1 1 1 1 0 0	7CH
C	0 0 1 1 1 0 0 1	39H
D	0 1 0 1 1 1 1 0	5EH
E	0 1 1 1 1 0 0 1	79H
F	0 1 1 1 0 0 0 1	71H

5.1.4 软件设计

从图 5-4 中可以看出 4 位七段数码管的 COM 端接在字位驱动上, 其他 8 个字段通过上拉排阻接在 V_{CC} 电源端, 利用 CPU 来控制数码管显示部分的导通和截止。

工作过程如下:

将要显示的字型代码送入字型锁存器锁存, 此时所有的数码管都可以显示同样的字符, 再将需要显示字型的地址送入字位锁存器锁存, 这样就可以在指定的字位上显示输入的字型。

1. 程序流程

单片机七段数码管动态显示程序的流程如图 5-5 所示。

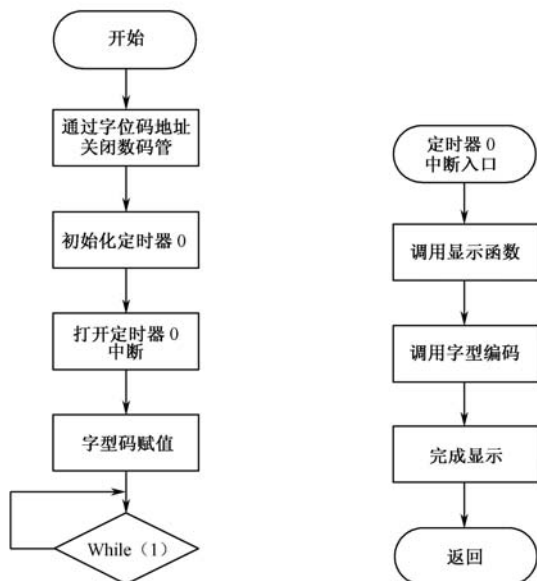


图 5-5 单片机七段数码管动态显示程序流程

2. 程序说明

单片机 7 段数码管动态显示程序代码及其说明如下：

```
#include <reg51.h>
#include <intrins.h>
#include <math.h>
unsigned char data dis_digit;           // dis_digit: 位选通值, 传送到 P2 口用于选通当
                                         //前数码管的数值, 如等于 0xfe 时,选通 P2.0 口数码管
unsigned char code dis_code[11]={0xc0,0xf9,0xa4,0xb0,           // 0, 1, 2, 3
                                0x99,0x92,0x82,0xf8,0x80,0x90, 0xff}; // 4, 5, 6, 7, 8, 9, off
unsigned char data dis_buf[8];          // dis_buf: 显于缓冲区基地址
unsigned char data dis_index;           // dis_index: 显示索引, 用于标识当前显示的数码
                                         //管和缓冲区的偏移量

Void delay(int t)
{
    int i;
    while(t--){
        for(i=0;i<120;i++)

        {;}
    }
}

Void main()
{
    P0 = 0xff;
    P2 = 0xff;
    TMOD = 0x01;           //定时器 0 工作于方式 1
    TH0 = 0x8C;
    TL0 = 0x8B;
    IE = 0x8A;             //开总中断和定时器 1 中断允许

    dis_buf[0] = dis_code[0x1];
    dis_buf[1] = dis_code[0x2];
    dis_buf[2] = dis_code[0x3];
    dis_buf[3] = dis_code[0x4];
    dis_buf[4] = dis_code[0x5];
    dis_buf[5] = dis_code[0x6];
    dis_buf[6] = dis_code[0x7];
    dis_buf[7] = dis_code[0x8];

    dis_digit = 0xf9;      //按位控制在数码管上显示的数字效果
```

```

dis_index = 0;

TR0 = 1;
while(1);

}

void timer0() interrupt 1
//定时器 0 中断服务程序, 用于数码管的动态扫描
//dis_index: 显示索引, 用于标识当前显示的数码管和缓冲区的偏移量
//dis_digit: 位选通值, 传送到 P2 口用于选通当前数码管的数值, 如等于 0xfe 时,
    选通 P2.0 口数码管
//dis_buf: 显于缓冲区基地址
{
    TH0 = 0x8c;
    TL0 = 0x8b;

    P2 = 0xff;                // 先关闭所有数码管
    P0 = dis_buf[dis_index];  // 显示代码传送到 P0 口
    P2 = dis_digit;

    dis_digit = _crol_(dis_digit,2);    // 位选通值左移, 下次中断时选通下一位数码管
    delay(500);
    dis_index++;

    dis_index &= 0x07;    // 8 个数码管全部扫描完一遍之后, 再回到第一个开始下一次扫描
}

```

5.2 单片机键盘程序（4×4 矩阵式）

键盘是单片机应用的一个重要方面，它可以实现输入数据、传送命令的功能，通过键盘可以实现“人机对话”，它是人工干预的主要手段。合理的键盘设计不仅可以节省系统的成本，更可使仪器设备的操作变得更为简单、方便，很大程度上提高系统综合性能。

根据其硬件的组成复杂程度可将键盘分为两大类：编码键盘和非编码键盘；根据其硬件连接的方式，又可将键盘分为独立连接式键盘和矩阵式键盘。

5.2.1 实例效果说明

图 5-6 是一个 4×4 矩阵式键盘的示意图，本例将实现的是 AT89S51 的并行口 P1 接 4×4 矩阵式键盘，以 P1.0~P1.3 作为输入线，以 P1.4~P1.7 作为输出线；在数码管上显示每个

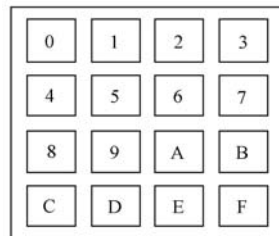


图 5-6 键盘示意图

按键的“0~F”序号，其对应的按键的序号排列如图 5-6 所示。

5.2.2 硬件电路设计

硬件设计是单片机开发的一个重要的部分，它构成了整个系统的实体，通过硬件部分的实验现象可以很容易地检验出软件设计是否正确合理。硬件调试是一种很好的软件程序检测方式。

1. 4×4 矩阵式键盘结构及工作原理分析

当按键数量较多时，通常将按键排列成矩阵形式，这样可以大大减少 I/O 口的占用，在矩阵式键盘中，每条水平线和垂直线在交叉处不直接连通，而是通过一个按键来连接。这样，一个端口就可以构成 4×4=16 个按键，比直接将端口线与键盘相连多出了 1 倍，如图 5-7 所示，而且线数越多，区别越明显，若再多加一条线就可以构成 20 键的键盘，而直接用端口线则只能多出一键（9 键）。

由此可见，在需要的键数比较多时，采用矩阵形式来做键盘是非常合理的。矩阵式结构的键盘显然比直接法要复杂一些，键盘识别相对也要复杂一些，列线通过电阻接正电源，并将行线所接的单片机的 I/O 口作为输出端，而列线所接的 I/O 口则作为输入。这样，当键没有被按下时，所有的输出端都是高电平，代表无键按下；行线输出是低电平，一旦有键按下，则输入线就会被拉低，这样，通过读入输入线的状态就可得知是否有键按下了。具体的识别及编程方法将在软件设计部分介绍。

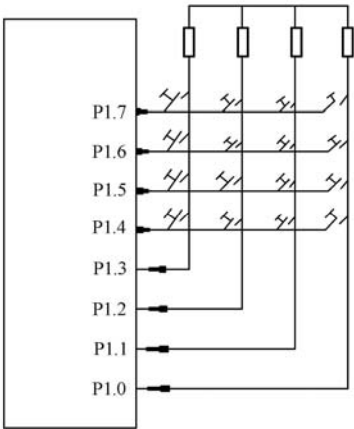


图 5-7 矩阵式键盘连线图

2. 硬件原理图

本例的硬件原理如图 5-8 所示。将单片机 P3.0~P3.7 口用 8 芯排线连接到“4×4 行列式键盘”区域中的 C1~C4、R1~R4 端口上；将单片机 P0.0/AD0~P0.7/AD7 端口用 8 芯排线连接到“四路静态数码显示模块”区域中的任何一个 a~h 端口上；要求 P0.0/AD0 对应着 a, P0.1/AD1 对应着 b, …，依次类推，P0.7/AD7 对应着 h。

5.2.3 软件程序设计

软件程序是这几部分的核心，因为只有完善的软件程序才能控制系统操作。相对来说，矩阵式键盘的软件设计比较复杂，其最终目的就是要实现以下几个功能：

- 是否有按键被按下；
- 消除抖动；
- 确定哪个按键被按下，若按键闭合了一次，操作也只能是一次。

在设计阶段，通信和自诊断的问题可以先放一边。一个扫描周期内还要完成的任务就是寄存输入的采样；执行程序并将结果进行存储；刷新寄存器。

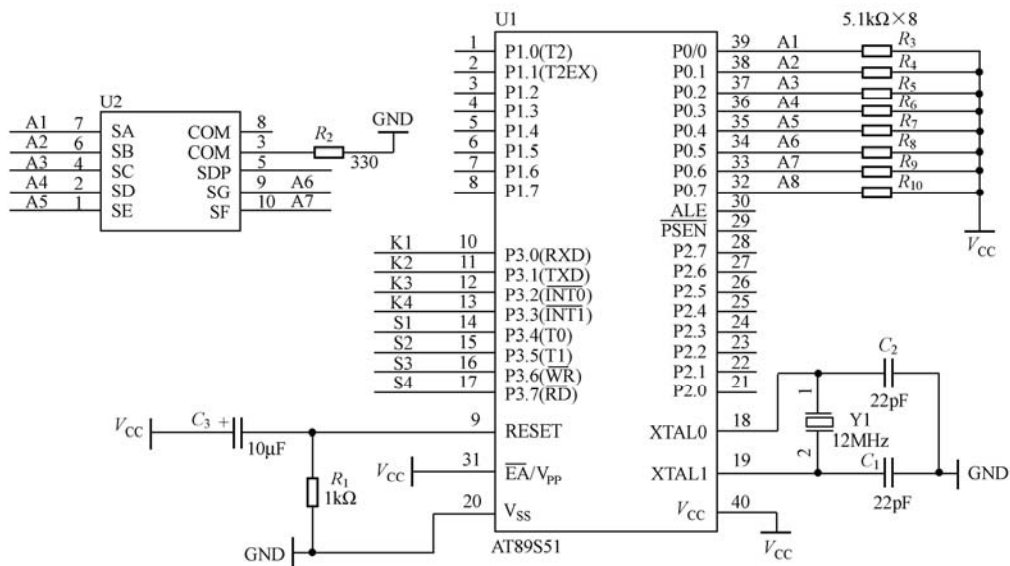


图 5-8 单片机控制矩阵式键盘的单片机部分电路图

1. 按键识别法

矩阵式键盘的按键识别方法是行扫描法。行扫描法又称为逐行（或列）扫描查询法，是一种最常用的按键识别方法，它可以准确地确定矩阵式键盘上哪个按键被按下。

其具体的扫描步骤如下：

① 判断键盘中有无按键按下。将全部行线置为低电平，然后检测列线的状态。只要列线中有一列为低电平，则表示键盘中有按键被按下，而且闭合的按键位于低电平线与 4 根行线交叉的 4 个按键之中。若所有列线均为高电平，则键盘中无按键按下。

② 判断闭合键所在的位置。在确认有按键按下后，即可进入确定具体闭合键的过程。其方法是依次将行线置为低电平，即在置某根行线为低电平时，其他线为高电平。在确定某根行线位置为低电平后，再逐行检测各列线的电平状态。若某列为低电平，则该列线与置为低电平的行线交叉处的按键就是闭合的按键。

2. 消除抖动

在软件设计部分除了要了解按键识别方法之外，还要解决到的一个问题就是消抖。键盘按键所用开关为机械弹性开关，利用了机械触点的合、断作用。由于机械触点的弹性作用，一个按键开关在闭合和断开的瞬间均有一连串的抖动。抖动时间的长短由按键的机械特性决定，一般为 5~10 ms，这是一个很重要的参数，在编写程序时这个时间必须要考虑在内。抖动过程引起电平信号的波动，有可能令 CPU 误解为多次按键操作，从而引起误处理，导致程序无法执行或错误执行。

为了确保 CPU 对一次按键动作只确认一次按键，必须消除抖动的影响。按键的消抖通常有软件和硬件两种消除方法。但是硬件消抖适用于按键的数目较少的情况，按键的数目较多的情况下常采用软件消抖的方法。

软件消抖通常采用软件延时的方法：在第一次检测到有按键按下时，执行一段延时的

子程序后，然后再确认电平是否仍保持闭合状态电平，如果保持闭合状态电平，则确认真正有按键按下，进行相应处理工作，从而消除了抖动的影响。这种消除抖动影响的软件方法是具有可操作性的。

3. 程序流程

单片机键盘程序流程如图 5-9 所示。

4. 行扫描流程图

行扫描流程如图 5-10 所示。

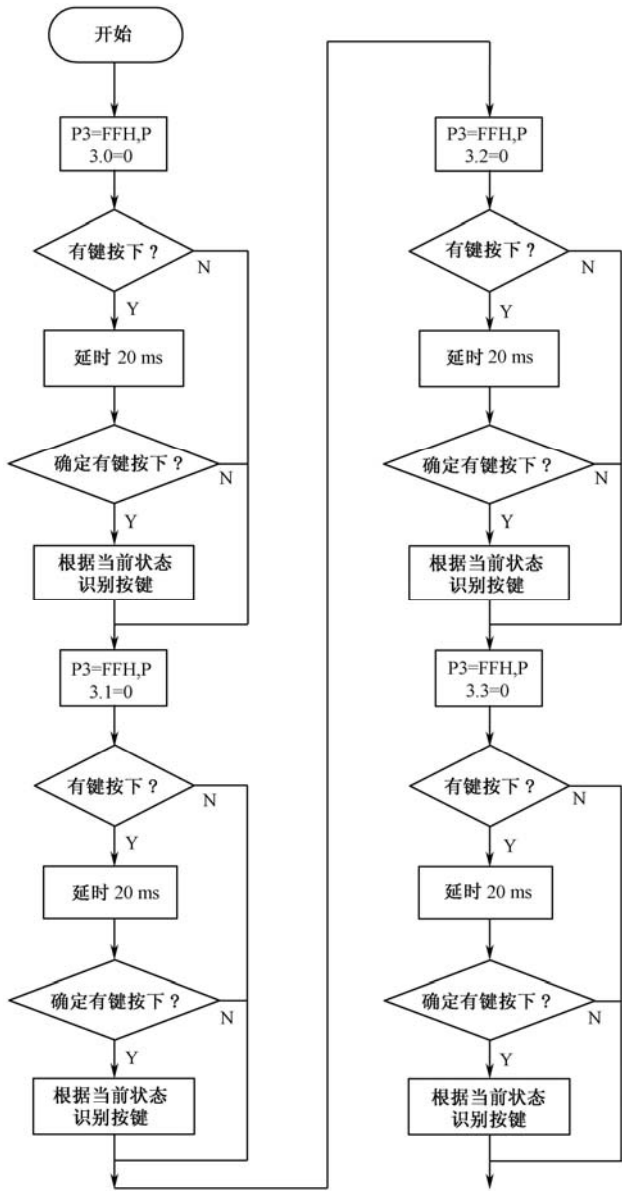


图 5-9 键盘程序流程图

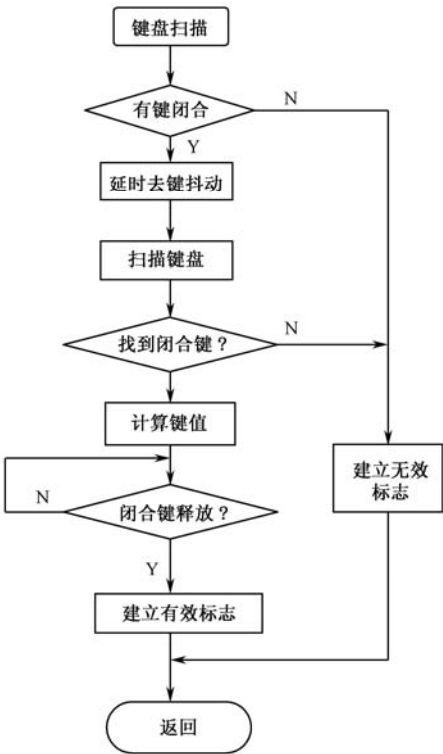


图 5-10 行扫描流程图

5. 程序说明

单片机键盘程序（4×4）如下：

```
#include <at89x51.h>

unsigned char code table[]={ 0x3f,0x06,0x5b,0x4f,
                             0x66,0x6d,0x7d,0x07,
                             0x7f,0x6f,0x77,0x7c,
                             0x39,0x5e,0x79,0x71 };

unsigned char temp;
unsigned char key;
unsigned char m,n;
/*主程序*/
void main(void)
{
    while(1)
    {
        P3=0xff;
        P3_4=0;
        temp=P3;
        temp=temp & 0x0f;           //取出低 4 位
        if (temp!=0x0f)             //有键按下
        {
            for(m=100;m>0;m--)
            for(n=200;n>0;n--);     //延时 20 ms
            temp=P3;
            temp=temp & 0x0f;       //取出低 4 位
            if (temp!=0x0f)         //有按键按下
            {
                temp=P3;
                temp=temp & 0x0f;   //取出低 4 位
                switch(temp)
                {
                    case 0x0e:
                        key=7;
                        break;
                    case 0x0d:
                        key=8;
                        break;
                    case 0x0b:
                        key=9;
                        break;
```

```

        case 0x07:
            key=10;
            break;
    }
    temp=P3;
    P1_0=~P1_0;
    P0=table[key];
    temp=temp & 0x0f;           //取出低 4 位
    while(temp!=0x0f)          //有按键按下
    {
        temp=P3;
        temp=temp & 0x0f;      //取出低 4 位
    }
}

P3=0xff;
P3_5=0;
temp=P3;
temp=temp & 0x0f;             //取出低 4 位
if (temp!=0x0f)               //有按键按下
{
    for(m=100;m>0;m--)
    for(n=200;n>0;n--);        //延时 20 ms
    temp=P3;
    temp=temp & 0x0f;           //取出低 4 位
    if (temp!=0x0f)             //有按键按下
    {
        temp=P3;
        temp=temp & 0x0f;      //取出低 4 位
        switch(temp)
        {
            case 0x0e:
                key=4;
                break;
            case 0x0d:
                key=5;
                break;
            case 0x0b:
                key=6;
                break;
            case 0x07:

```



```

        key=11;
        break;
    }
    temp=P3;
    P1_0=~P1_0;
    P0=table[key];
    temp=temp & 0x0f;    //取出低 4 位
    while(temp!=0x0f)    //有按键按下
    {
        temp=P3;
        temp=temp & 0x0f;    //取出低 4 位
    }
}

P3=0xff;
P3_6=0;
temp=P3;
temp=temp & 0x0f;    //取出低 4 位
if (temp!=0x0f)    //有按键按下
{
    for(m=100;m>0;m--)
    for(n=200;n>0;n--);    //延时 20 ms
    temp=P3;
    temp=temp & 0x0f;    //取出低 4 位
    if (temp!=0x0f)    //有按键按下
    {
        temp=P3;
        temp=temp & 0x0f;    //取出低 4 位
        switch(temp)
        {
            case 0x0e:
                key=1;
                break;
            case 0x0d:
                key=2;
                break;
            case 0x0b:
                key=3;
                break;
            case 0x07:
                key=12;

```

```

        break;
    }
    temp=P3;
    P1_0=~P1_0;
    P0=table[key];
    temp=temp & 0x0f;        //取出低 4 位
    while(temp!=0x0f)        //有按键按下
    {
        temp=P3;
        temp=temp & 0x0f;    //取出低 4 位
    }
}

P3=0xff;
P3_7=0;
temp=P3;
temp=temp & 0x0f;          //取出低 4 位
if (temp!=0x0f)            //有按键按下
{
    for(m=100;m>0;m--)
    for(n=200;n>0;n--);    //延时 20 ms
    temp=P3;
    temp=temp & 0x0f;      //取出低 4 位
    if (temp!=0x0f)        //有按键按下
    {
        temp=P3;
        temp=temp & 0x0f;  //取出低 4 位
        switch(temp)
        {
            case 0x0e:
                key=0;
                break;
            case 0x0d:
                key=13;
                break;
            case 0x0b:
                key=14;
                break;
            case 0x07:
                key=15;
                break;

```

```

    }
    temp=P3;
    P1_0=~P1_0;
    P0=table[key];
    temp=temp & 0x0f;    //取出低 4 位
    while(temp!=0x0f)    //有按键按下
    {
        temp=P3;
        temp=temp & 0x0f;    //取出低 4 位
    }
    }
}
}
}

```

5.3 单片机控制LCD显示

随着电子行业的飞速发展，单片机可选择的外部设备也越来越多，本例将介绍用单片机控制 LCD 显示。LCD (liquid crystal display)，中文称为液晶平面显示器或液晶显示器，它的工作原理是利用液晶的物理特性，通电时排列变得有序，使光线容易通过；不通电时排列混乱，阻止光线通过。

5.3.1 实例说明

本例将编写一个简单的程序，在 LCD 显示器上显示出汉字和图形。通过这个实例来了解 LCD 显示汉字和图形的方法，以及学习使用 Keil C51 进行单片机开发 LED 显示器的方法。

5.3.2 芯片介绍

本实例采用的是型号为 12232-4 的图形点阵液晶显示器，它主要由行驱动器/列驱动器及 122×32 全点阵液晶显示器组，可完成图形显示，也可以显示七个半（16×16 点阵）汉字。

1. 12232-4 的主要性能及内部结构

12232-4 的主要性能如下：

- 电源： $V_{DD}=+5V$ ；
- 显示内容：122（列）×32（行）点；
- 显示颜色：绿底蓝字；
- 显示角度：6 点钟直视；
- STN 正视反射模式；
- 驱动方式：1/32 Duty, 1/6 Bias；

- 工作温度：0~+60℃，存储温度：-10~+70℃；
- 连接方式：外部接口由带缆连接。

12232-4 的内部原理如图 5-11 所示

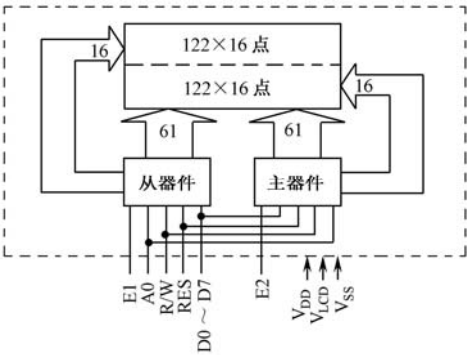


图 5-11 12232-4 的内部原理图

2. 12232-4 的引脚说明

12232-4 的引脚说明如表 5-2 所示。

表 5-2 12232-4 的引脚说明

引脚号	引脚名称	取值范围	引脚功能描述
1	V _{DD}	+5V	电源电压
2	V _{SS}	0V	电源地
3	V _{LCD}	0~+5V	LCD 外接驱动负电压
4	RES	H/L	复位信号
5	E1	H/L	读/写使能信号（MASTER）
6	E2	H/L	读/写使能信号（SLAVE）
7	R/W	H/L	读/写选择信号
8	D/I	H/L	D/I=“H”，表示 DB7~DB0 为显示数据 D/I=“L”，表示 DB7~DB0 为显示指令数据
9	DB0	H/L	数据线
10	DB1	H/L	数据线
11	DB2	H/L	数据线
12	DB3	H/L	数据线
13	DB4	H/L	数据线
14	DB5	H/L	数据线
15	DB6	H/L	数据线
16	DB7	H/L	数据线
17	NC	—	悬空
18	NC	—	悬空

3. 常用指令描述

(1) 显示模式设置

CODE:

A0	$\overline{\text{RD}}$	$\overline{\text{WR}}$	D7	D6	D5	D4	D3	D2	D1	D0
L	H	L	H	L	H	L	H	H	H	D

功能：控制屏幕显示的开关，不改变显示 RAM（DD RAM）中的内容，同样也不影响内部状态。当 D=0 时，表示打开显示；当 D=1 时，表示关闭显示。如果在显示关闭的状态下选择静态驱动模式，那么内部电路将处于安全模式。

(2) 设置显示起始行

CODE:

A0	$\overline{\text{RD}}$	$\overline{\text{WR}}$	D7	D6	D5	D4	D3	D2	D1	D0
L	H	L	H	H	L	A4	A3	A2	A1	A0

功能：确定哪一行显示在屏幕的第一行。起始地址可以是 0~31 范围内的任意一行，行地址计数器用于显示行扫描同步，具有循环计数的功能，当扫描完一行后就自动加 1。

(3) 页地址设置

CODE:

A0	$\overline{\text{RD}}$	$\overline{\text{WR}}$	D7	D6	D5	D4	D3	D2	D1	D0
L	H	L	H	L	H	H	H	L	A1	A0

功能：设置页地址。在 CPU 对 DD RAM 进行读/写操作之前，首先要设置页地址和列地址，该设置不影响显示。

A1	A0	页地址
0	0	0
0	1	1
1	0	2
1	1	3

(4) 列地址设置

CODE:

A0	$\overline{\text{RD}}$	$\overline{\text{WR}}$	D7	D6	D5	D4	D3	D2	D1	D0
L	H	L	L	A6	A5	A4	A3	A2	A1	A0

功能：设置 DD RAM 中的列地址。在 CPU 对 DD RAM 进行读/写操作之前，首先要设置页地址和列地址。执行读/写命令后，列地址会自动加 1，达到 50 时停止，但页地址不变。

A6	A5	A4	A3	A2	A1	A0	列地址
0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1
1	0	0	1	1	1	0	4E
1	0	0	1	1	1	1	4F

5.3.3 硬件设计

图 5-12 是 12232-4 芯片与 C51 单片机的接口电路原理图，图中 UA1、SRD、SWR、UAD0~UAD7 和 RESET 引脚分别与单片机相连，LCD1 和 LCD2 用于调试亮度。

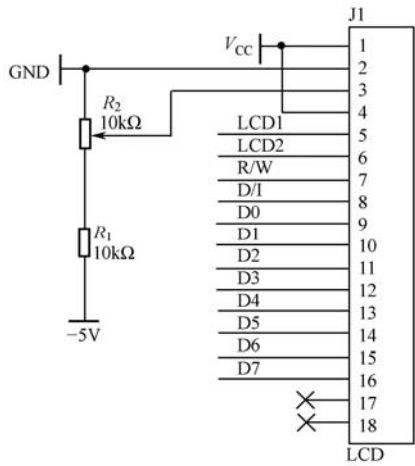


图 5-12 硬件原理图

5.3.4 软件设计

软件设计包括开始、初始化 LCD、清除 LCD、写 LCD 等几个过程。

1. 程序流程图

程序设计的流程如图 5-13 所示。

2. 程序代码

本例的程序代码如下：

```
/*LCD 显示*/
/*AT89C51, 12 MHz*/
#include<reg51.h>
#include<intrins.h>
#define LCD P0
#define uchar unsigned char
sbit A0 = P1^4;
sbit _WR = P1^3; //定义 WR BIT(PC5) 读 1/写 0
sbit E1 = P1^1; //定义 E1 BIT(PC4) 片选 1 (主窗口)
sbit E2 = P1^2; //定义 E2 BIT(PC7) 片选 2 (从窗口)
uchar dzhc[32]; //点阵缓存区
uchar sxhc[4]; //RAM 数据显示缓存区
```



图 5-13 程序流程图

```

void start(void);           //初始化 LCD
void clear(void);          //LCD 清屏
void wait_ready(void);     //等待 ready
void display(uchar col,uchar layer,uchar width,uchar *bmp);  //点阵码显示输出
void disasc(uchar col,uchar layer,uchar ascii_code,uchar mode);
//单个 ASCII 码输出(ascii_code 为 ASCII 编码)
void disdata(uchar col,uchar layer,uchar n,uchar mode);      //RAM 数据（数字）显示输出
void dprintf(uchar col,uchar layer,uchar *buf,uchar mode);    //通用混合字符串显示
typedef struct recieve    //汉字字模数据结构
{
    uchar shouyin[2];
    uchar model[32];
};
/****************************/
uchar code ascii[] =
{
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x20,0xF8,0xF8,0x20,0xF8,0xF8,0x20,0x00,//#
    0x02,0x0F,0x0F,0x02,0x0F,0x0F,0x02,0x00,
    0x38,0x7C,0x44,0x47,0x47,0xCC,0x98,0x00,$
    0x03,0x06,0x04,0x1C,0x1C,0x07,0x03,0x00,
    0x30,0x30,0x00,0x80,0xC0,0x60,0x30,0x00,/%
    0x0C,0x06,0x03,0x01,0x00,0x0C,0x0C,0x00,
    0x80,0xD8,0x7C,0xE4,0xBC,0xD8,0x40,0x00,/&
    0x07,0x0F,0x08,0x08,0x07,0x0F,0x08,0x00,
    0x00,0x10,0x1E,0x0E,0x00,0x00,0x00,0x00,/"
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0xF0,0xF8,0x0C,0x04,0x00,0x00,/"
    0x00,0x00,0x03,0x07,0x0C,0x08,0x00,0x00,
    0x00,0x00,0x04,0x0C,0xF8,0xF0,0x00,0x00,/"
    0x00,0x00,0x08,0x0C,0x07,0x03,0x00,0x00,
    0x80,0xA0,0xE0,0xC0,0xC0,0xE0,0xA0,0x80,///
    0x00,0x02,0x03,0x01,0x01,0x03,0x02,0x00,
    0x00,0x80,0x80,0xE0,0xE0,0x80,0x80,0x00,/+
    0x00,0x00,0x00,0x03,0x03,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,/"
    0x00,0x00,0x10,0x1E,0x0E,0x00,0x00,0x00,
    0x80,0x80,0x80,0x80,0x80,0x80,0x80,0x00,/-
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,/"
    0x00,0x00,0x00,0x0C,0x0C,0x00,0x00,0x00,
    0x00,0x00,0x00,0x80,0xC0,0x60,0x30,0x00,///

```

0x0C,0x06,0x03,0x01,0x00,0x00,0x00,0x00,
0xF8,0xFC,0x04,0xC4,0x24,0xFC,0xF8,0x00,//0
0x07,0x0F,0x09,0x08,0x08,0x0F,0x07,0x00,
0x00,0x10,0x18,0xFC,0xFC,0x00,0x00,0x00,//1
0x00,0x08,0x08,0x0F,0x0F,0x08,0x08,0x00,
0x08,0x0C,0x84,0xC4,0x64,0x3C,0x18,0x00,//2
0x0E,0x0F,0x09,0x08,0x08,0x0C,0x0C,0x00,
0x08,0x0C,0x44,0x44,0x44,0xFC,0xB8,0x00,//3
0x04,0x0C,0x08,0x08,0x08,0x0F,0x07,0x00,
0xC0,0xE0,0xB0,0x98,0xFC,0xFC,0x80,0x00,//4
0x00,0x00,0x00,0x08,0x0F,0x0F,0x08,0x00,
0x7C,0x7C,0x44,0x44,0xC4,0xC4,0x84,0x00,//5
0x04,0x0C,0x08,0x08,0x08,0x0F,0x07,0x00,
0xF0,0xF8,0x4C,0x44,0x44,0xC0,0x80,0x00,//6
0x07,0x0F,0x08,0x08,0x08,0x0F,0x07,0x00,
0x0C,0x0C,0x04,0x84,0xC4,0x7C,0x3C,0x00,//7
0x00,0x00,0x0F,0x0F,0x00,0x00,0x00,0x00,
0xB8,0xFC,0x44,0x44,0x44,0xFC,0xB8,0x00,//8
0x07,0x0F,0x08,0x08,0x08,0x0F,0x07,0x00,
0x38,0x7C,0x44,0x44,0x44,0xFC,0xF8,0x00,//9
0x00,0x08,0x08,0x08,0x0C,0x07,0x03,0x00,
0x00,0x00,0x00,0x30,0x30,0x00,0x00,0x00,//:
0x00,0x00,0x00,0x06,0x06,0x00,0x00,0x00,
0x00,0x00,0x00,0x30,0x30,0x00,0x00,0x00,//;
0x00,0x00,0x08,0x0E,0x06,0x00,0x00,0x00,
0x00,0x80,0xC0,0x60,0x30,0x18,0x08,0x00,//<
0x00,0x00,0x01,0x03,0x06,0x0C,0x08,0x00,
0x40,0x40,0x40,0x40,0x40,0x40,0x40,0x00,//=
0x02,0x02,0x02,0x02,0x02,0x02,0x02,0x00,
0x00,0x08,0x18,0x30,0x60,0xC0,0x80,0x00,//>
0x00,0x08,0x0C,0x06,0x03,0x01,0x00,0x00,
0x18,0x1C,0x04,0xC4,0xE4,0x3C,0x18,0x00,//?
0x00,0x00,0x00,0x0D,0x0D,0x00,0x00,0x00,
0xF0,0xF8,0x08,0xC8,0xC8,0xF8,0xF0,0x00,//@
0x07,0x0F,0x08,0x0B,0x0B,0x0B,0x01,0x00,
0xE0,0xF0,0x98,0x8C,0x98,0xF0,0xE0,0x00,//A
0x0F,0x0F,0x00,0x00,0x00,0x0F,0x0F,0x00,
0x04,0xFC,0xFC,0x44,0x44,0xFC,0xB8,0x00,//B
0x08,0x0F,0x0F,0x08,0x08,0x0F,0x07,0x00,
0xF0,0xF8,0x0C,0x04,0x04,0x0C,0x18,0x00,//C
0x03,0x07,0x0C,0x08,0x08,0x0C,0x06,0x00,

0x04,0xFC,0xFC,0x04,0x0C,0xF8,0xF0,0x00, //D
 0x08,0x0F,0x0F,0x08,0x0C,0x07,0x03,0x00,
 0x04,0xFC,0xFC,0x44,0xE4,0x0C,0x1C,0x00, //E
 0x08,0x0F,0x0F,0x08,0x08,0x0C,0x0E,0x00,
 0x04,0xFC,0xFC,0x44,0xE4,0x0C,0x1C,0x00, //F
 0x08,0x0F,0x0F,0x08,0x00,0x00,0x00,0x00,
 0xF0,0xF8,0x0C,0x84,0x84,0x8C,0x98,0x00, //G
 0x03,0x07,0x0C,0x08,0x08,0x07,0x0F,0x00,
 0xFC,0xFC,0x40,0x40,0x40,0xFC,0xFC,0x00, //H
 0x0F,0x0F,0x00,0x00,0x00,0x0F,0x0F,0x00,
 0x00,0x00,0x04,0xFC,0xFC,0x04,0x00,0x00, //I
 0x00,0x00,0x08,0x0F,0x0F,0x08,0x00,0x00,
 0x00,0x00,0x00,0x04,0xFC,0xFC,0x04,0x00, //J
 0x07,0x0F,0x08,0x08,0x0F,0x07,0x00,0x00,
 0x04,0xFC,0xFC,0xC0,0xF0,0x3C,0x0C,0x00, //K
 0x08,0x0F,0x0F,0x00,0x01,0x0F,0x0E,0x00,
 0x04,0xFC,0xFC,0x04,0x00,0x00,0x00,0x00, //L
 0x08,0x0F,0x0F,0x08,0x08,0x0C,0x0E,0x00,
 0xFC,0xFC,0x38,0x70,0x38,0xFC,0xFC,0x00, //M
 0x0F,0x0F,0x00,0x00,0x00,0x0F,0x0F,0x00,
 0xFC,0xFC,0x38,0x70,0xE0,0xFC,0xFC,0x00, //N
 0x0F,0x0F,0x00,0x00,0x00,0x0F,0x0F,0x00,
 0xF0,0xF8,0x0C,0x04,0x0C,0xF8,0xF0,0x00, //O
 0x03,0x07,0x0C,0x08,0x0C,0x07,0x03,0x00,
 0x04,0xFC,0xFC,0x44,0x44,0x7C,0x38,0x00, //P
 0x08,0x0F,0x0F,0x08,0x00,0x00,0x00,0x00,
 0xF8,0xFC,0x04,0x04,0x04,0xFC,0xF8,0x00, //Q
 0x07,0x0F,0x08,0x0E,0x3C,0x3F,0x27,0x00,
 0x04,0xFC,0xFC,0x44,0xC4,0xFC,0x38,0x00, //R
 0x08,0x0F,0x0F,0x00,0x00,0x0F,0x0F,0x00,
 0x18,0x3C,0x64,0x44,0xC4,0x9C,0x18,0x00, //S
 0x06,0x0E,0x08,0x08,0x08,0x0F,0x07,0x00,
 0x00,0x1C,0x0C,0xFC,0xFC,0x0C,0x1C,0x00, //T
 0x00,0x00,0x08,0x0F,0x0F,0x08,0x00,0x00,
 0xFC,0xFC,0x00,0x00,0x00,0xFC,0xFC,0x00, //U
 0x07,0x0F,0x08,0x08,0x08,0x0F,0x07,0x00,
 0xFC,0xFC,0x00,0x00,0x00,0xFC,0xFC,0x00, //V
 0x01,0x03,0x06,0x0C,0x06,0x03,0x01,0x00,
 0xFC,0xFC,0x00,0x80,0x00,0xFC,0xFC,0x00, //W
 0x03,0x0F,0x0E,0x03,0x0E,0x0F,0x03,0x00,
 0x0C,0x3C,0xF0,0xC0,0xF0,0x3C,0x0C,0x00, //X

```

0x0C,0x0F,0x03,0x00,0x03,0x0F,0x0C,0x00,
0x00,0x3C,0x7C,0xC0,0xC0,0x7C,0x3C,0x00,//Y
0x00,0x00,0x08,0x0F,0x0F,0x08,0x00,0x00,
0x1C,0x0C,0x84,0xC4,0x64,0x3C,0x1C,0x00,//Z
0x0E,0x0F,0x09,0x08,0x08,0x0C,0x0E,0x00,
0x80,0x80,0x80,0x80,0xe0,0xC0,0x80,0x00,    //->0x5b (自定义显示字符)
0x01,0x01,0x01,0x01,0x07,0x03,0x01,0x00

```

```
};
```

```
/*液晶显示控制命令表*/
```

```
struct typFNT_GB16 code GB_16[] =          // 数据表
```

```
{
```

```

"单", 0x00,0x00,0xF8,0x49,0x4A,0x4C,0x48,0xF8,
      0x48,0x4C,0x4A,0x49,0xFC,0x08,0x00,0x00,
      0x10,0x10,0x17,0x12,0x12,0x12,0x12,0xFF,
      0x12,0x12,0x12,0x12,0x13,0x18,0x10,0x00,

```

```

"片", 0x00,0x00,0xFE,0x20,0x20,0x20,0x20,0x20,
      0x3F,0x20,0x20,0x20,0x20,0x30,0x20,0x00,
      0x80,0x40,0x3F,0x01,0x01,0x01,0x01,0x01,
      0x01,0xFF,0x00,0x00,0x00,0x00,0x00,0x00,

```

```

"机", 0x10,0x10,0xD0,0xFF,0x90,0x10,0x00,0xFC,
      0x04,0x04,0x04,0xFE,0x04,0x00,0x00,0x00,
      0x04,0x03,0x00,0xFF,0x80,0x41,0x20,0x1F,
      0x00,0x00,0x00,0x3F,0x40,0x40,0x70,0x00

```

```
};
```

```
#define off          0xAE          //显示关闭
```

```
#define disp_on      0xAF          //显示打开
```

```
#define disp_start_line 0xC0          //显示起始地址（后 5 位表示 0~31 行）
```

```
#define page_addr_set 0xB8          //页地址设置（0~3）
```

```
#define col_addr_set  0x00          //列地址设置（0~61）
```

```
#define status_busy   0x80          //0=ready
```

```
#define mode_write     0xEE          //写模式
```

```
#define dynamic_driver 0xA4          //动态驱动
```

```
#define adc_select      0xA0          //clockwise
```

```
#define clk32          0xA9          //刷新时钟设置 1/32
```

```
#define clk16          0xA8          //刷新时钟设置 1/16
```

```
#define reset          0xE2          //软件复位
```

//新的驱动着重在简化代码，这样外部可调用的函数为：系统初始化，清屏，判别标志

//ASCII 和汉字混合输出函数，RAM 缓冲数据显示输出，一般用于输出数字

```
#define uchar unsigned char
```

```

#define uint    unsigned int

void clear();

void main(void)
{
    uchar a; uint b;
    for(a=0;a<10;a++)
        for(b=1;b<10;b++);
    /*开机延时*/
    start();
    clear();
    dprintf(10,1,"Wo Ai DanPianJi",0); ///在 up 行从第 10 列开始显示/正常显示
    disdata(10,0,2,1);          ///在 down 行从第 10 列开始显示/反白显示
    while(1);
}

void send1(uchar c)
{
    E1=0;
    E2=1;
    wait_ready();
    A0=1;          //数据
    _WR=0;         //写触发
    LCD =c;
    _nop_();
    E2=0;
}

//
void send2(uchar c)
{
    E2=0;          //关 S
    E1=1;          //开 M
    wait_ready();
    A0=1;
    _WR=0;         //写触发
    _nop_();
    LCD = c;
    _nop_();
    E1=0;          //关 M
}

//
void send3(uchar instruction)
{
    E1=0;          //关 M

```

```

        E2=1;                //开 S
        wait_ready();
        A0=0;                //指令
        _WR=0;               //写触发
        _nop_();
        LCD=instruction;     //指令码
        _nop_();
        E2=0;                //关 S
    }
//
void send4(uchar instruction)
{
    E2=0;                    //关 S
    E1=1;                    //开 M
    wait_ready();
    A0=0;                    //指令
    _WR=0;                   //写触发
    _nop_();
    LCD= instruction;       //指令码
    _nop_();
    E1=0;                    //关 M
}
//等待 ready:等待 LCD 内部操作完成
void wait_ready(void)
{
    A0=0;                    //指令
    _WR=1;                   //读
    _nop_();
    while(LCD & status_busy);
    //读入 LCD 状态, 1=忙, 一直等待 LCD 内部操作完成 DDRA = 0xff; 重新置 PA 口输出
}
//12232-4LCD 初始化, 开机后仅调用一次
void start(void)
{
    send4(reset);            //复位, m-left,s-right
    send3(reset);
    send4(off);              //关闭显示
    send3(off);
    send4(dynamic_driver);   //动态驱动
    send3(dynamic_driver);
    send4(clk32);            //1/32 占空比

```

```

send3(clk32);
send4(adc_select);           //顺时针方向
send3(adc_select);
send4(mode_write);          //写模式
send3(mode_write);
send4(col_addr_set);
send4(disg_start_line);      //归回零列，设定显示起始行首
send3(col_addr_set);
send3(disg_start_line);
send4(disg_on);              //开显示
send3(disg_on);
}
//清屏
void clear(void)
{
    uchar i, page;
    for (page=0;page<4;page++)
    {
        send4(page_addr_set|page);
        send3(page_addr_set|page);
        send4(0);             //主窗口设置为 0 列
        send3(0);             //从窗口设置为 0 列
        for (i=0;i<62;i++)
        {
            send2(0x00);
            send1(0x00);
        }
    }
}
//同时设置主（右）从（左）显示页为 0~3 页
void set_page(uchar page)
{
    send4(page_addr_set|page);
    send3(page_addr_set|page);
}
//8 同时设置主（右）从（左）列地址为 0~61 列
void set_address(uchar address)
{
    send4(address&0x7F);       //&0x7F，防止超限
    send3(address&0x7F);
}

```

//在右页（从窗口）当前地址画 1 B

```
void putchar_l(uchar c)
{
    send2(c);
}
```

//在左页（主窗口）当前地址画 1 B

```
void putchar_r(uchar c)
{
    send1(c);
}
```

void display(uchar col,uchar layer,uchar width,uchar *bmp)

```
{    uchar x;
    uchar address;           //address 表示显存的物理地址
    uchar p=0;
    uchar page=0;
    uchar window=0;         //page 表示上下两页，window 表示左右窗口（0 左，1 右）
    if (layer) page=2;       //左——主窗口，右——从窗口
    for (x=col; x<col+width; x++)
    {
        if (x>121)
            return;          //防止显示乱码
        if (x>60)            //左右窗口定位
        {
            window=1;        //右——从窗口
            address=x%61;     }

        else
            address=x;        //主窗口输出

        set_page(page);      //上层数据输出
        set_address(address);

        if (window)
            putchar_r(bmp[p]);

        else
            putchar_l(bmp[p]);

        set_page(page+1);    //下层数据输出
        set_address(address); //列保持不变

        if (window)
            putchar_r(bmp[p+width]);
    }
}
```

```

        else
        putchar_l(bmp[p+width]);
        p++;    }
    }
    //显示单个 ASCII 码 col——列; layer——上下行: 1——上, 0——下; ascii_code:所要显示的
    //ASCII 码。ASCII (8×16) 和汉字 (16×16) 显示函数
    void dprintf(uchar col,uchar layer,uchar *ptr,uchar mode)
    {
        uchar c1,c2;
        uchar m,n,k;
        uchar ulen;
        uchar ucol,ulayer;
        ulen = 0;
        ucol = col;
        ulayer = layer;

        while (ptr[ulen] != 0) ulen++; //探测字串长度
        m=0;
        while(m<ulen)
        {
            c1 = ptr[m];
            c2 = ptr[m+1];
        }
        //ASCII 字符与汉字内码的区别在于 128 做分界, 大于 128 的为汉字码
        if(c1 <=128)
        //ASCII
        {
            if(mode)disasc(ucol,ulayer,c1,1);
            else
                disasc(ucol,ulayer,c1,0);
            ucol+=8;
            m++; //ASCII 码的处理
        }
        else
        //中文
        {
            for(n=0;n<sizeof(word16)/sizeof(word16[0]);n++)
            {
                //查找定位当前汉字的点阵区

                if((c1 == word16[n].shouyin[0]) && (c2 == word16[n].shouyin[1]))
                    break;
            }

```

```

        for(k=0;k<32;k++)
    { if(mode)dzhc[k]=~word16[m].model[k];
      else
        dzhc[k]= word16[m].model[k];

    }
    display(ucol,ulayer,16,dzhc);
    ucol+=16;
    m+=2;          //中文的处理

  }
}

void disasc(uchar col,uchar layer,uchar ascii_code,uchar mode)
{
    uchar k;
    for(k=0;k<16;k++)      //ASCII 码显示占用 16 B
    {
        if(mode)dzhc[k]=~ascii[(ascii_code-0x20)*16 + k];
        else
            dzhc[k]= ascii[(ascii_code-0x20)*16 + k];

    }

    display(col,layer,8,dzhc); } //显示小于 4 个的十进制数字，修改缓冲区的大小可以扩展显示

void disdata(uchar col,uchar layer,uchar n,uchar mode)
{
    while(n--)
    {
        if(mode)
            disasc(col,layer,sxhc[n]+0x30,1);
        else
            disasc(col,layer,sxhc[n]+0x30,0);
        col += 8;
    }
}

```

5.4 带有存储功能的数显温度计

随着人们生活水平的不断提高，单片机控制无疑是人们追求的目标之一，它所给人带来的方便也是不可否认的，其中数显温度计就是一个典型的例子，但人们对它的要求越来越

高，要为现代人工作、科研、生活提供更好的、更方便的设施就需要从数单片机技术入手，一切向着数字化控制、智能化控制方向发展。

5.4.1 实例说明

日常生活中使用的温度计大多数是利用水银柱的高度指示刻度，随着科技的发展，利用电子计算后得出的数字由显示器件直观地观察到，这种温度计称为数显温度计。数显温度计是一种电子产品，由感温元件（如温度电阻，它会随温度变化，阻值也会变化）来识别温度，模/数转换电路（把阻值变化转变成电信号），识别放大，驱动显示屏发光，显示出温度。

5.4.2 芯片介绍

本例采用了美国 Dallas 公司生产的一种功能较强的数字式温度传感器 DS1624 芯片，它集测量系统和存储器于一体，它比同系列的 DS1620 芯片控制更为简单，比 DS1621 芯片分辨率更高，可以使用一片控制器控制多达 8 片传感器，数字接口电路简单，与 I²C 总线兼容。DS1624 芯片的测温范围宽，数字温度输出达 13 位、读数稳定、分辨率高，精度为 0.03125℃，可在最低 2.7V 电压下工作，适用于低功耗应用系统，无需外接电路，与单片机接口简单，可以广泛用于温度检测、温度控制和温度报警等领域。

1. DS1624 工作性能

- 无需外围元件即可测量温度；
- 测量温度值以 13 位数字量输出（双字节传输）；
- 测量温度的典型转换时间小于 1 s；
- 测量范围为-55~+125℃，精度为 0.03125℃；
- 内部集成了 256 B 的 E²PROM，可以用来保存用户设定的参数；
- 数据的读/写通过 2 线串行接口实现（SDA、SCL）可选总线地址；
- 采用 8 引脚 DIP 或 SOIC 封装。

2. 功能结构

DS1624 芯片的内部功能结构如图 5-14 所示。

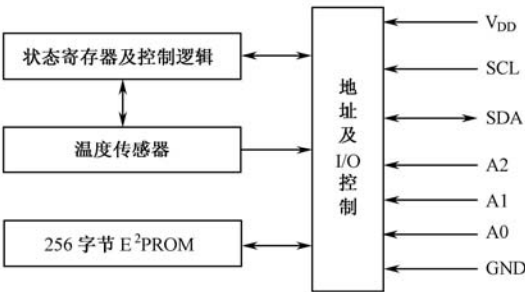


图 5-14 内部功能结构图

3. 引脚分布和定义

DS1624 芯片的引脚分布如图 5-15 所示，图 5-15（a）是 SOIC 封装，图 5-15（b）是 DIP 封装。

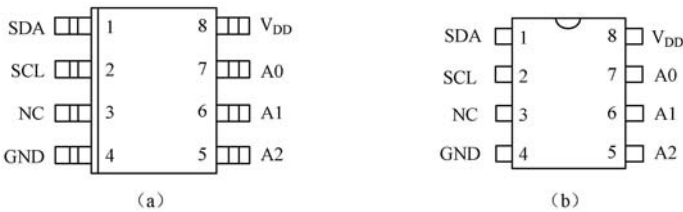


图 5-15 DS1624 引脚分布

引脚描述及功能如表 5-3 所示。

表 5-3 DS1624 引脚功能描述

引 脚 名 称	说 明
SDA	2 线 I ² C 串行数据输入/输出
SCL	2 线 I ² C 串行时钟端
NC	未连接
GND	电源地
A2	片选地址输入 A2
A1	片选地址输入 A1
A0	片选地址输入 A0
V _{DD}	电源端 (+2.7~5.5 V)

4. DS1624 工作原理

DS1624 芯片采用了专用的片内温度测量技术进行温度的测量，在计数门开通的情况下，对低温系数振荡器的脉冲个数进行计数，计数脉冲的周期由高温系数振荡器所决定，在计数器和温度寄存器中预置了一个初值，它相当于-55℃。如果计数周期结束之前计数器达到 0，已预置了此初值的温度寄存器中的数字就会增加，从而表明温度高于-55℃。同时计数器斜坡累加电路被重新预置一个值，并重新开始计数，如果脉冲周期在计数器到 0 之前还未结束，则重复上面的过程，否则停止计数。最终温度寄存器中的值即为被测温度值。通过改变增加的每 1℃内的计数器的计数，斜坡累加电路可以补偿振荡器的非线性误差，以提高精度，任意温度下计数器的值和每一斜坡累加电路的值对应的计数次数必须为已知。

温度测量的原理结构如图 5-16 所示。

DS1624 芯片的温度值的精度为 0.03125℃，内部的温度寄存器为 13 位（2 B），在发出读温度值请求后还会输出两位补偿值。该寄存器可以通过 I²C 总线串行读出，高位（MSB）在前，低位（LSB）在后。该 13 位寄存器的内容即为补码表示的温度值，最高位置符号位，符号位为 1 表示温度值为负，为 0 表示温度值为正。将该 13 位数据的真值乘以 0.03125，即为被测温度值。测量的温度与输出数据的关系如表 5-4 所示。

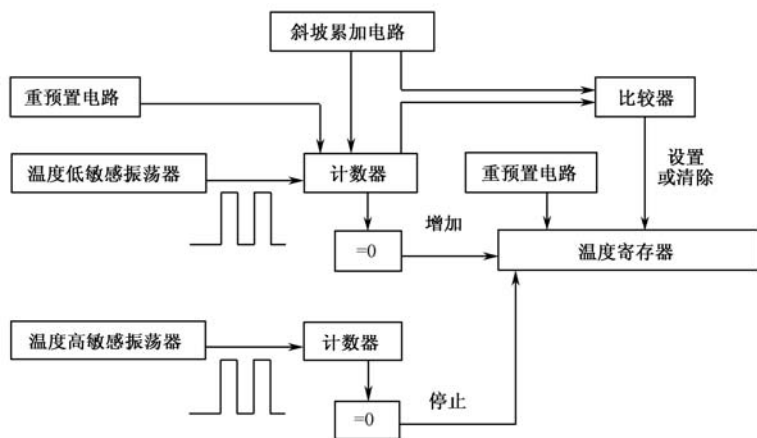


图 5-16 温度测量的原理结构图

表 5-4 温度与输出数据关系表

测 量 温 度	数字量输出（二进制）	数字量输出（十六进制）
+125℃	0111 1101 0000 0000	7D00H
+25.0625℃	0001 1001 0001 0000	1910H
+0.5℃	0000 0000 1000 0000	0080H
+0℃	0000 0000 0000 0000	0000H
-0.5℃	1111 1111 1000 0000	FF80H
-25.0625℃	1110 0110 1111 0000	E6F0H
-55℃	1100 1001 0000 0000	C900H

由于数据在 I^2C 总线上传输时高位在前，所以 DS1624 芯片读出的数据可以是两个字节，也可以是一个字节（分辨率为 $1^{\circ}C$ ）。两个字节中的第二个字节包含的最低位为 $0.03125^{\circ}C$ 。

13 位温度寄存器中存储温度值的数据格式如表 5-5 所示。

表 5-5 温度值的数据存储格式

S	B14	B13	B12	B11	B10	B9	B8		B7	B6	B5	B4	B3	0	0	0
高 8 位字节									低 8 位字节							

其中 S 为符号位，当 $S=1$ 时，表示当前的测量的温度为负的温度；当 $S=0$ 时，表示当前的测量的温度为正的温度；B14~B3 为当前测量的温度值，最低 3 位被设置为 0。

5. DS1624 工作方式

DS1624 芯片可以工作在两种方式下：连续转换方式和一次转换方式。使用哪种工作模式是由片上的配置/状态寄存器来决定的，配置/状态寄存器定义如表 5-6 所示。

表 5-6 配置/状态寄存器格式

DONE	1	0	0	1	0	1	1SHOT
------	---	---	---	---	---	---	-------

其中 1SHOT 为温度转换模式选择。1SHOT=1 时为一次转换模式，DS1624 在收到启动温度转换命令 EEH 后进行一次温度转换；1SHOT=0 时为连续转换模式，此时 DS1624 将继续进行温度转换，并将最近一次的结果保存在温度寄存器中，该位为非易失性的。DONE 为转换完成位，温度正在进行转换时为 0，转换完成时为 1。

6. 存储器操作

控制器对 DS1624 的存储器编程有页编程模式、字节编程模式、读模式三种模式。

(1) 页编程模式

在页编程模式中，如同字节写方式一样，先将控制字节、访问存储器指令（17H）、字地址发送到 DS1624，接着发 N 个数据字节，其中以 8 个字节为一个页面。主器件发送不多于一个页面字节的数据字节到 DS1624，这些数据字节暂存在片内页面缓存器中，在主器件发送停止信号以后写入到存储器。接收到每一个字节以后，低位顺序地址指针在内部加 1。高位顺序字地址保持为常数。如果主器件在产生停止条件以前要发送多于一页字的数据，地址计数器将会循环，并且先接收到的数据将被覆盖。像字节写操作一样，一旦停止条件被接收到，则内部写周期将重新开始。

(2) 字节编程模式

在字节编程模式中，主控制器发送地址和一个字节的数据到 DS1624。在主器件发出开始（START）信号以后，主器件发送写控制字节，即 1001A2A1A00（其中 R/W 控制位为低电平）。指示从接收器寻址，DS1624 接收后应答，再由主器件发送访问存储器指令（17H），DS1624 接收后应答，接着由主器件发送的下一个字节字地址将被写入到 DS1624 的地址指针。主器件接收到来自 DS1624 的另一个确认信号以后，发送数据字节，并写入到寻址的存储地址。DS1624 再次发出确认信号，同时主器件产生停止条件 STOP，启动内部写周期，在内部写周期 DS1624 将不产生确认信号。

(3) 存储器的读模式

在这种模式下，主器件可以从 DS1624 的 E²PROM 中读取数据。主器件在发送开始信号之后，主器件首先发送写控制字节 1001A2A1A00，主器件接收到 DS1624 应答之后，发送访问存储器的指令（17H），收到 DS1624 的应答之后，接着发送字地址将被写入到 DS1624 的地址指针。

这时 DS1624 发送应答信号之后，主器件并没有发送停止信号，而是重新发送 START 信号，接着又发送读控制字节 1001A2A1A01，主器件接收到 DS1624 应答之后，开始接收 DS1624 发送来的数据，主器件每接收完一个字节的数据之后，都要发送一个应答信号给 DS1624，直到主器件发送一个非应答信号或停止条件来结束 DS1624 的数据发送过程。

7. DS1624 的指令集

在写信息操作时，主器件输出从器件（DS1624）的地址，同时 R/W 位置 0，接收到响应位后，总线上的主器件发出一个命令地址，DS1624 接收此地址后，产生响应位，主器件就向它发送数据。

主机对 DS1624 写操作流程如图 5-17 所示，主机对 DS1624 读操作流程如图 5-18 所示。



图 5-17 主机对 DS1624 写操作流程图



图 5-18 主机对 DS1624 读操作流程图

如果要对从器件进行读操作，主器件除了发出命令地址外，还要产生一个重复的启动条件和命令字节，此时 R/W 位置 1，读操作开始。

下面对它们的命令进行说明。

① 开始温度转换指令[EEH]：该指令用于启动温度转换，无需再输入数据。在一次转换模式下，该指令启动转换，DS1624 完成转换之后保持空闲；在连续转换方式下，该指令启动 DS1624 进行连续的温度转换。

② 访问存储器指令[17H]：该指令用来访问 DS1624 内部集成的 256 B 的 E²PROM，发送该指令之后，下一个字节就是被访问存储器的字地址数据，即可进行 E²PROM 的读/写操作。

③ 访问设置寄存器指令[ACH]：若 R/W=0，该数据写入配置寄存器之后，MCU 送出一个字节，用以确定 DS1624 的工作方式；若 R/W=1，DS6124 将读出存在寄存器中的值，并通知 MCU 转换是否完成。

④ 读温度值指令[A AH]：该指令读出最后一个温度转换的结果，随后 DS1624 将两个字节补码表示的温度值送出。最高为符号位，最低 3 位不用。

⑤ 温度转换结束指令[22H]：该指令停止温度转换，无需再输入数据。该指令停止 DS1624 的温度转换，并且保持空闲，直到 DS1624 得到新的温度转换开始指令。

8. DS1624 工作时序

DS1624 在嵌入一个系统时，需要有 MCU 对其发出控制命令，如读/写状态寄存器、读温度寄存器、开始温度转换等命令，MCU 对 DS1624 的控制是通过 I²C 总线接口来实现的，写入和读出完全遵循 I²C 总线的协议。

DS1624 的工作时序如图 5-19 所示。

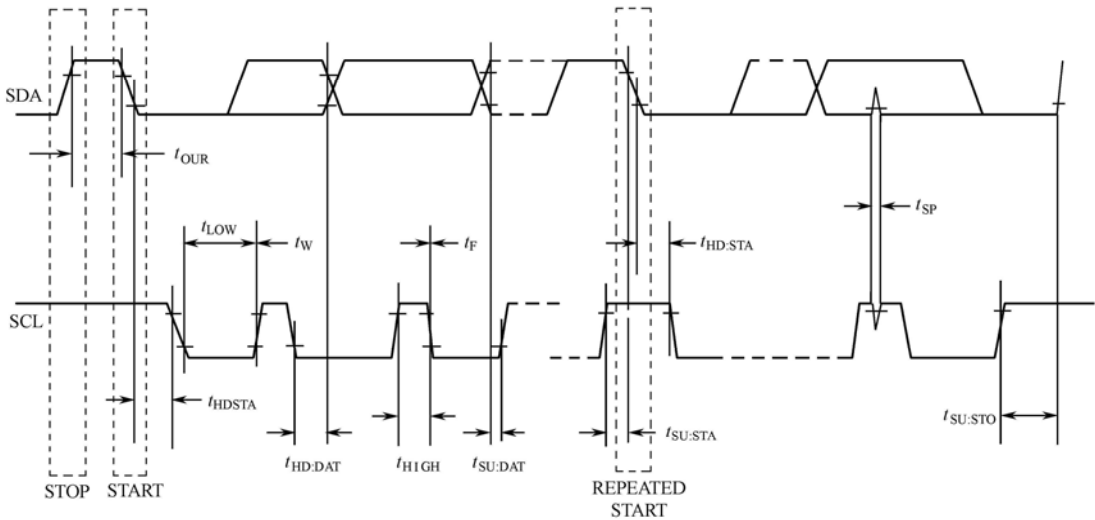


图 5-19 DS1624 的工作时序图

DS1624 的 SDA 线并没有针对 SCL 进行内部延迟。因此，写入数据时，在 SCL 转变为低电平前，SDA 的逻辑电平必须在 DS1624 外部被保持，否则会被误认为是启动或停止信号。在 2 线总线上写逻辑 1 时，在 SDA 电平降低到安全的逻辑高域值电平 V_{IH} ($0.7 \times V_{DD}$ ，最小) 前，必须保证 SCL 电平已经低于逻辑低域值电平 V_{IL} ($0.3 \times V_{DD}$ ，最大)。写逻辑 0 时，在 SDA 电平上升到 V_{IL} 前，SCL 电平必须已经低于 V_{IL} 。产生 START 信号时，在 SCL 降低到 V_{IH} 前 SDA 必须已经低于 V_{IL} 。产生 STOP 信号时，SCL 降低到 V_{IH} 前 SDA 必须已经高于 V_{IH} 。每个器件的 V_{IL} 和 V_{IH} 值都进行了生产测试，以保证在整个电压和温度范围内，即使存在器件制造容差，这个时序都能正确运行。

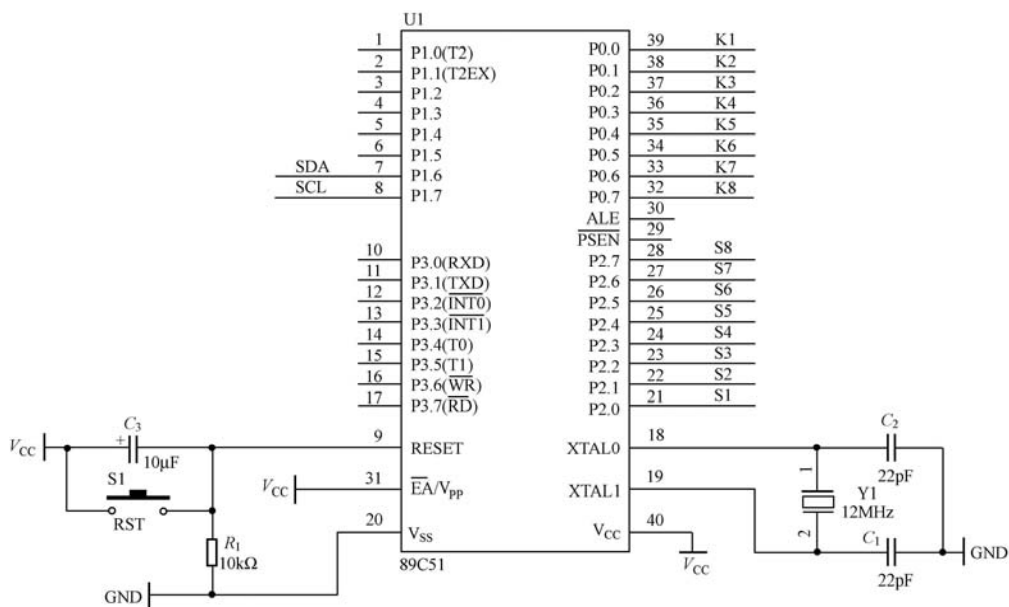
DS1624 在使用的时候，应注意几点问题：

- ① 写 E²PROM 需要 10 ms 的时间，所以在每一次寄存器写操作后都需要等待 10ms 再进行下一次写操作。
- ② 由于 SDA、SCL 均为漏极开路 I/O 口，因此一定要有上拉电阻。
- ③ 在数据传输和写命令字的时候，一定遵循 I²C 总线的协议，在由写操作到读操作转换的时候，应该重新启动数据传输，然后发送地址和读/写位。
- ④ 在构成测温系统时，一片单片机最多可以连接 8 片 DS1624，并可采用求平均值的方法提高测量精度，此时应将地址 A2、A1、A0 进行不同调整。

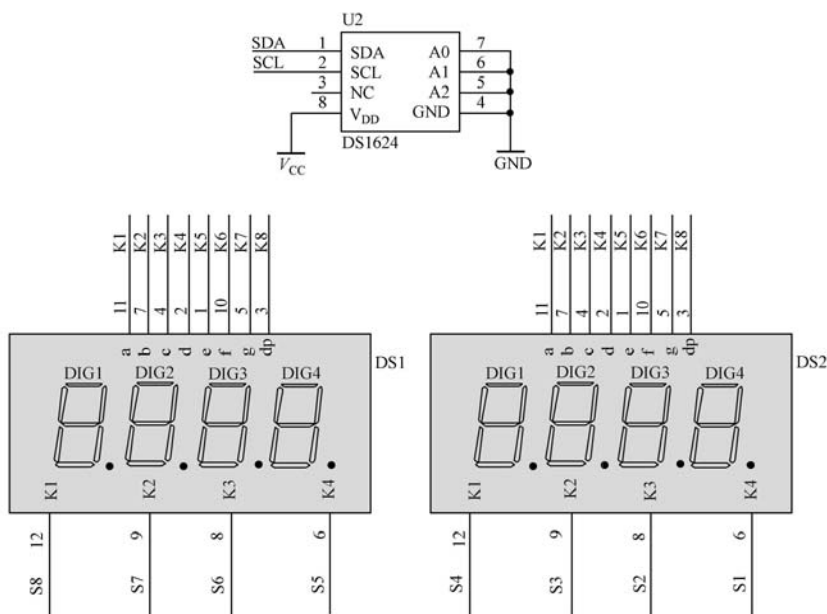
5.4.3 硬件电路设计

本例用一片 DS1624 完成本地数字温度的测量，并通过 8 位数码管显示出测量到的温度值，其硬件电路如图 5-20 所示。

由图中可以看出，单片机的 P0.0~P0.7 口连接到动态数码的 a、b、c、d、e、f、g、h 上；P2.0~P2.7 口连接到动态数码的 S1、S2、S3、S4、S5、S6、S7、S8 上；单片机的 P1.6 口连接串行数据输入/输出口 (SDA)；P1.7 口连接串行时钟端 (SCL)。



(a) 单片机控制部分



(b) 温度控制与显示部分

图 5-20 带有存储功能的数显温度计

5.4.4 软件设计

由于 DS1624 是 I²C 总线结构的串行数据传送，它只需要 SDA 和 SCL 两根线就能完成数据的传送过程。因此在进行程序设计时，必须按照 I²C 的协议来对 DS1624 芯片数据访问。

1. 程序流程

要从 DS1624 中读取温度值，应首先启动 DS1624 的内部温度 A/D 转换器，有相应的命

令用来启动开始温度转换。对 DS1624 的读/写操作时要严格遵循 I²C 总线协议，利用单片机可以仿真 I²C 总线的读/写时序，I²C 总线的控制主要包括开始、读数据、写数据、应答、停止五部分。

DS1624 的初始化子程序、读取温度值和启动温度转换子程序流程如图 5-21 所示。

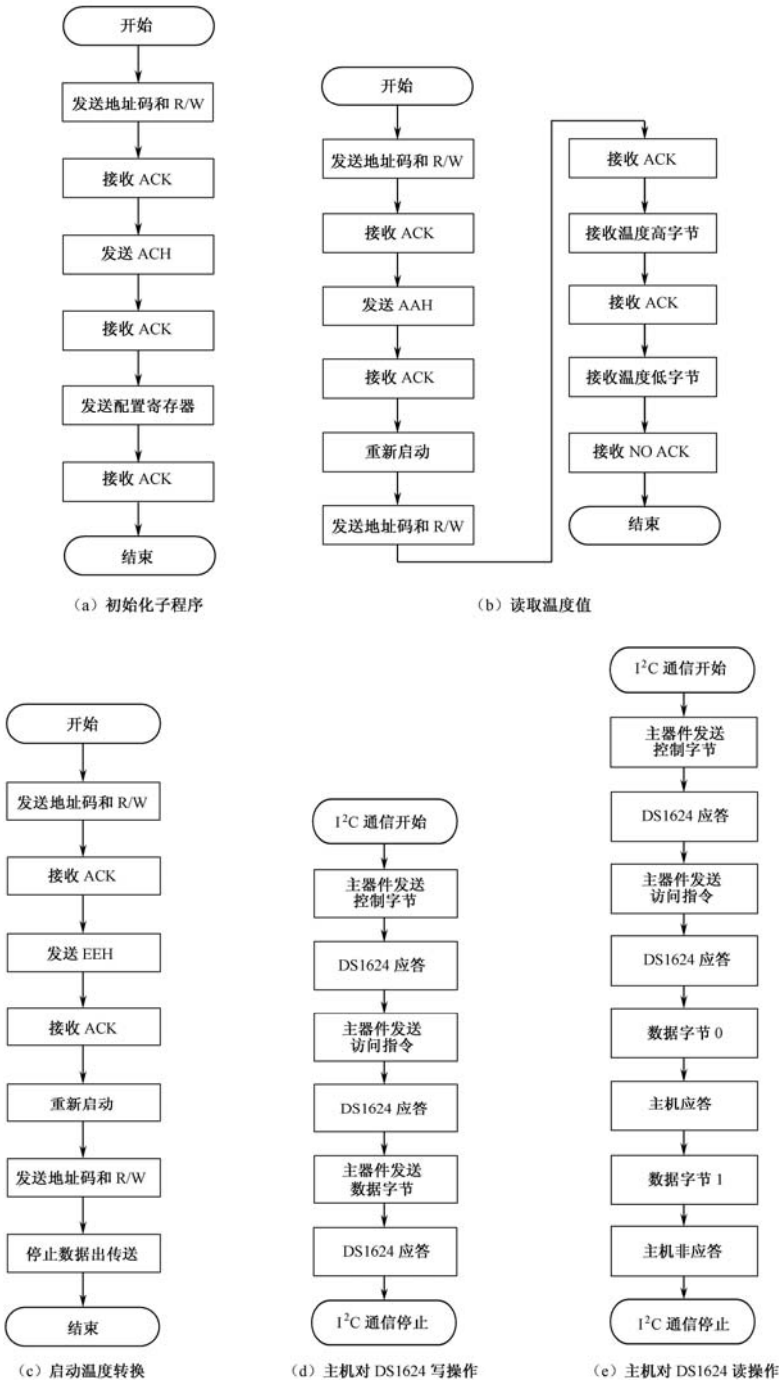


图 5-21 程序流程图

2. 程序说明

```
#include <at89x52.h>
#include <intrins.h>

#define uchar unsigned char
#define uint unsigned int

uchar code displaybit[]={0xfe,0xfd,0xfb,0xf7,
                        0xef,0xdf,0xbf,0x7f};           //显示位
uchar code displaycode[]={0x3f,0x06,0x5b,0x4f,
                        0x66,0x6d,0x7d,0x07,
                        0x7f,0x6f,0x77,0x7c,
                        0x39,0x5e,0x79,0x71,0x00};      //显示码
uchar code dotcode[32]={0,3,6,9,12,16,19,22,
                        25,28,31,34,38,41,44,48,
                        50,53,56,59,63,66,69,72,
                        75,78,81,84,88,91,94,97};
sbit SDA=P1^6;    //I2C 串行数据输入/输出
sbit SCL=P1^7;    //I2C 串行时钟
uchar displaybuffer[8]={0,1,2,3,4,5,6,7};             //显示缓冲
uchar eepromdata[8];                                  //数据存储
uchar temperdata[2];
uchar timecount;                                       //定时器计数器
uchar displaycount;                                    //显示计数器
bit secondflag=0;                                       //秒标志位
uchar secondcount=0;                                    //秒计数器
uchar retn;
uint result;
uchar x;
uint k;
uint ks;
void delay(void);                                       //延时函数
void delay10ms(void);                                   //延时 10 s 函数
void i_start(void);                                     //开始函数
void i_stop(void);                                      //停止函数
void i_init(void);                                      //初始化函数
void i_ack(void);                                       //命令正确应答函数
bit i_clock(void);                                      //时钟函数
bit i_send(uchar i_data);                              //发送函数
uchar i_receive(void);                                  //接收函数
```

```

bit start_temperature_T(void);           //启动温度计函数
bit read_temperature_T(uchar *p);       //读取温度计函数
//延时函数
void delay(void)
{
    _nop_();
    _nop_();
    _nop_();
    _nop_();
    _nop_();
}
//延时 10 ms 函数
void delay10ms(void)
{
    uint i;
    for(i=0;i<1000;i++)
    {
        delay();
    }
}
//开始函数
void i_start(void)
{
    SCL=1;           //I2C 串行时钟端使能
    delay();
    SDA=0;
    delay();
    SCL=0;           //I2C 串行时钟端禁止
    delay();
}
//停止函数
void i_stop(void)
{
    SDA=0;
    delay();
    SCL=1;
    delay();
    SDA=1;
    delay();
    SCL=0;
    delay();
}

```

```

//初始化函数
void i_init(void)
{
    SCL=0;
    i_stop();
}
//应答函数
void i_ack(void)
{
    SDA=0;
    i_clock();
    SDA=1;
}
//时钟函数
bit i_clock(void)
{
    bit sample;
    SCL=1;
    delay();
    sample=SDA;
    _nop_();
    _nop_();
    SCL=0;
    delay();
    return(sample);
}
//发送函数
bit i_send(uchar i_data)
{
    uchar i;
    for(i=0;i<8;i++)
    {
        SDA=(bit)(i_data & 0x80);
        i_data=i_data<<1;
        i_clock();
    }
    SDA=1;
    return(~i_clock());
}
//接收函数
uchar i_receive(void)

```

```

{
    uchar i_data=0;
    uchar i;
    for(i=0;i<8;i++)
    {
        i_data*=2;
        if(i_clock()) i_data++;
    }
    return(i_data);
}

//启动温度计函数
bit start_temperature_T(void)
{
    i_start();
    if(i_send(0x90))
    {
        if(i_send(0xee))
        {
            i_stop();
            delay();
            return(1);
        }
        else
        {
            i_stop();
            delay();
            return(0);
        }
    }
    else
    {
        i_stop();
        delay();
        return(0);
    }
}

//读温度计函数
bit read_temperature_T(uchar *p)
{
    i_start();
    if(i_send(0x90))

```

```

{
    if(i_send(0xaa))
    {
        i_start();
        if(i_send(0x91))
        {
            *(p+1)=i_receive();
            i_ack();
            *p=i_receive();
            i_stop();
            delay();
            return(1);
        }
        else
        {
            i_stop();
            delay();
            return(0);
        }
    }
    else
    {
        i_stop();
        delay();
        return(0);
    }
}
else
{
    i_stop();
    delay();
    return(0);
}
}

void main( )
{
    P1=0xff;
    timecount=0;
    displaycount=0;
    TMOD=0x21;           //设置定时器工作方式
    TH1=0x06;            //定时器计数
    TL1=0x06;

```

```

TR1=1;                //启动定时器 1
ET1=1;                //开定时器 1 中断
ET0=1;                //开定时器 0
EA=1;
if(start_temperature_T()) //向 DS1624 发送启动 A/D 温度转换命令，成功则启动 T0 定时 1 s
{
    secondflag=0;
    secondcount=0;
    TH0=55536/256;
    TL0=55536%6;
    TR0=1;
}
while(1)
{
    if(secondflag==1)
    {
        secondflag=0;
        TR0=0;                //启动定时器 0
        if(read_temperature_T(temperdata)) //T0 定时 1 s 时间到，读取 DS1624 的温度值
        {
            for(x=0;x<8;x++)
            {
                displaybuffer[x]=16;
            }
            x=2;
            result=temperdata[1]; //将读取的温度值进行数据处理，并送到显示缓冲区
            while(result/10)
            {
                displaybuffer[x]=result%10;
                result=result/10;
                x++;
            }
            displaybuffer[x]=result;
            result=temperdata[0];
            result=result>>3;
            displaybuffer[0]=(dotcode[result])%10;
            displaybuffer[1]=(dotcode[result])/10;
            if(start_temperature_T()) //温度值数据处理完毕，重新启动 DS1624 开始温度转换
            {
                secondflag=0;
                secondcount=0;
                TH0=55536/256;
            }
        }
    }
}

```

```

        TL0=55536%6;
        TR0=1;
    }
}
}
}
}
//定时器 0 中断服务子程序
void t0(void) interrupt 1 using 0           //T0 用于定时 1 s 时间
{
    secondcount++;
    if(secondcount==100)
    {
        secondcount=0;
        secondflag=1;
    }
    TH0=55536/256;
    TL0=55536%6;
}
void t1(void) interrupt 3 using 0           //T1 定时 1 ms，用于数码管的动态刷新
{
    timecount++;
    if(timecount==4)                       //T1 定时 1 ms
    {
        timecount=0;
        if (displaycount==5)
        {
            P0=(displaycode[displaybuffer[7-displaycount]] | 0x80); //在该位同时还要显示小数点
        }
        else
        {
            P0=displaycode[displaybuffer[7-displaycount]];
        }
        P2=displaybit[displaycount];
        displaycount++;
        if(displaycount==8)
        {
            displaycount=0;
        }
    }
}
}

```

5.5 单片机实现数字电压表显示

在电气测量中，电压是一个很重要的参数。如何准确地测量模拟信号的电压值，一直是电测仪器研究的内容之一。数字电压表是诸多数字化仪表的核心与基础，电压表的数字化是将连续的模拟量，如直流电压转换成不连续的离散的数字量并加以显示，这有别于传统的以指针加刻度盘进行读数的方法，避免了读数的视差和视觉疲劳。目前数字万用表的内部核心部件是 A/D 转换器，转换器的精度很高。

在现代检测技术中，常需用高精度数字电压表进行现场检测，将检测到的数据送入微计算机系统，完成计算、存储、控制和显示等功能。数字电压表是通用仪器中使用较广泛的一种测试仪器，很多电量或非电量经变化后都用可数字电压表完成测试。因此，数字电压表被广泛地应用于科研和生产测试中。

5.5.1 实例说明

数字电压表对繁多的电量测试具有精度高、测量速度快、自动化程度高等优点，在科研生产的电量测试中得到了广泛的应用。

本例中数字电压表的控制系统采用 AT89C52 单片机，A/D 转换器采用 ADC0809 为主要硬件，数字电压表测量 0~5 V 的 8 路输入电压值，并在 4 位 LED 数码管上轮流显示或单路选择显示。该系统的数字电压表电路简单，所用的元件较少，成本低，调节工作可实现自动化。还可以方便地进行 8 路 A/D 转换量的测量，远程测量结果传送等功能。

5.5.2 设计思路分析

本例将介绍一种以单片机为核心的电压测量仪表，它能够测量电压量，并且测量结果能够通过数码管显示，从而具有一定的智能性。

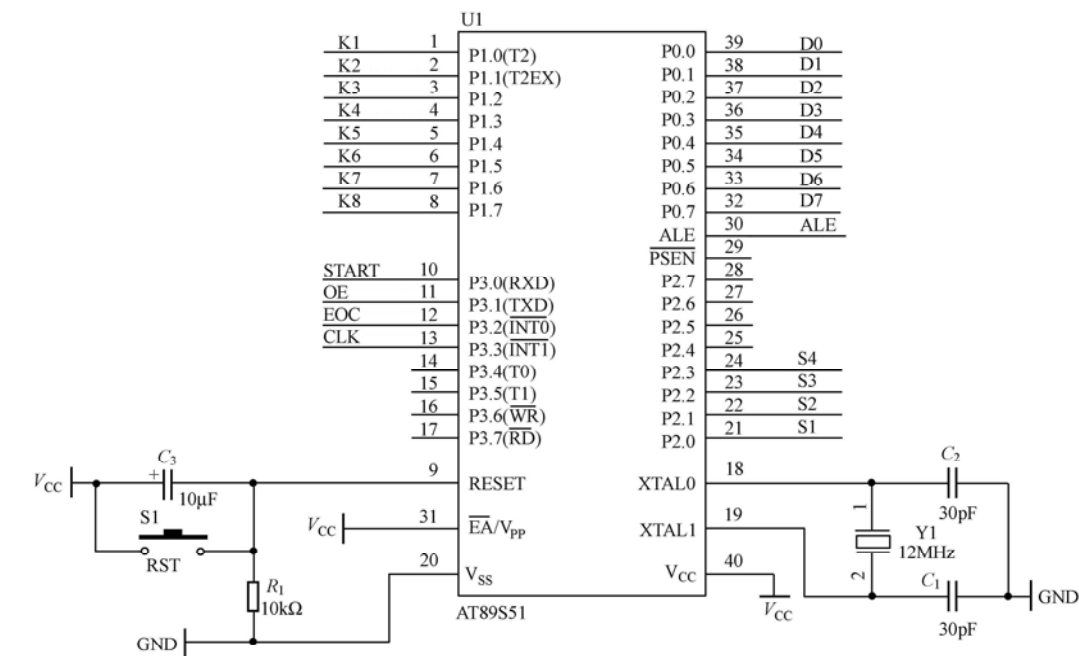
由于单片机的有效输入/输出信号均为数字信号，而对于整个系统的前向通道有效信号均应为模拟信号，所以在设计过程中必然包括模拟量转换为数字量的单元设计。根据要求，本例采用 ADC0809 芯片作为转换电路。

硬件部分的设计主要任务就是对电压信号能够进行测量并显示；在软件部分，主要是各个模块电路的软件设计，能够将采集到的模拟量转换为数字量，并显示。

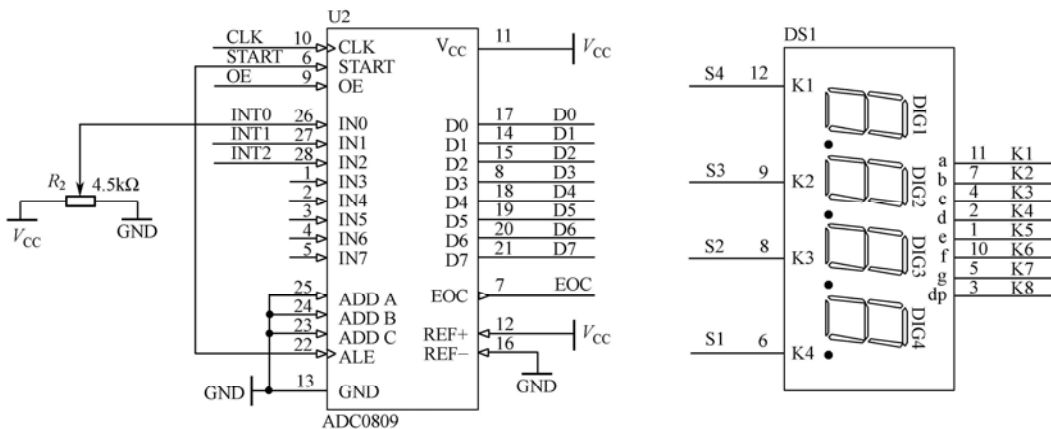
5.5.3 硬件电路设计

ADC0809 对输入模拟量的要求是：信号单极性，电压范围为 0~5V，若信号太小，必须进行放大；输入的模拟量在转换过程中应该保持不变，如果模拟量变化太快，则需在输入前增加采样保持电路。单片机实现数字电压表显示的电路连接如图 5-22 所示。

把单片机中的 P1.0~P1.7 口通过 8 芯排线与动态数码显示区域中的 A、B、C、D、E、F、G、H 端口相连接；把单片机中的 P2.0~P2.3 口通过 8 芯排线与动态数码显示区域中的 S1、S2、S3、S4 端口相连接；把单片机中的 P3.0 口与模/数转换模块中的 ST 脚相连接；把单片机中的 P3.1 口与模/数转换模块中的 OE 脚相连接；把单片机中的 P3.2 口与模/数转换模块中的 EOC 脚相连接；把单片机中的 P3.3 口与模/数转换模块中的 CLK 脚相连接。



(a) 单片机控制部分



(b) 模/数转换与实现部分

图 5-22 单片机实现数字电压表显示的电路图

把模/数转换模块中的 A2、A1、A0 脚连接到电源模块中的 GND 脚上；把模/数转换模块中的 IN0 脚连接到三路可调电压模块中的 VR1 脚上。

把单片机中的 P0.0~P0.7 口用 8 芯排线连接到模/数转换模块区域中的 D0、D1、D2、D3、D4、D5、D6、D7 脚上。

5.5.4 软件设计

由于 ADC0809 在进行 A/D 转换时需要有 CLK 信号，而此时的 ADC0809 的 CLK 是接在 AT89S51 单片机的 P3.3 口上，也就是要求从 P3.3 口输出 CLK 信号供 ADC0809 使用。因

此产生 CLK 信号的方法就得用软件来产生了。

1. 设计流程

单片机实现数字电压表显示的软件设计流程如图 5-23 所示，它的软件滤波子函数流程图如图 5-24 所示。

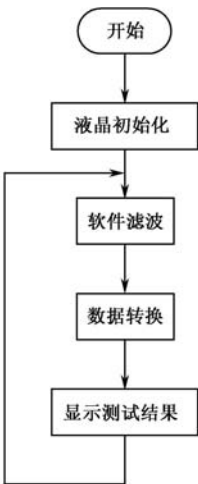


图 5-23 主程序流程图

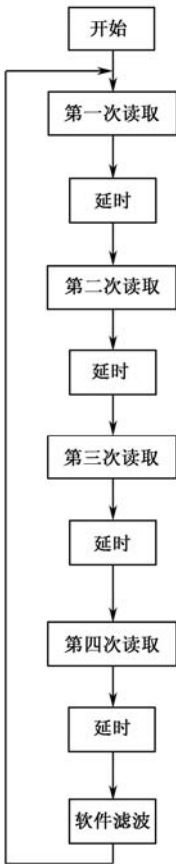


图 5-24 软件滤波子程序流程图

2. 程序说明

单片机实现数字电压表显示的程序说明如下：

```
#include <at89x52.h>

#define uchar unsigned char
#define uint unsigned int

uchar code dispbitcode[]={0xfe,0xfd,0xfb,0xf7,
                           0xef,0xdf,0xbf,0x7f};           //显示位
uchar code dispcode[]={0x3f,0x06,0x5b,0x4f,0x66,
```

0x6d,0x7d,0x07,0x7f,0x6f,0x00}); //显示码

```
uchar dispbuf[8]={10,10,10,10,0,0,0,0}; //显示缓冲
uchar dispcount; //显示计数器
uchar getdata;
uint temp;
uchar i;
sbit ST=P3^0; // A/D 转换启动信号
sbit OE=P3^1; //数据输出允许信号
sbit EOC=P3^2; // A/D 转换结束信号
sbit CLK=P3^3; //时钟脉冲输入端
void main(void)
{
    ST=0;
    OE=0;
    ET0=1; //打开定时器 0 中断
    ET1=1; //打开定时器 1 中断
    EA=1;
    TMOD=0x12; //设置工作方式
    TH0=216; //定时器 0 定时
    TL0=216;
    TH1=(65536-4000)/256; //定时器 1 定时
    TL1=(65536-4000)%256;
    TR1=1; //启动定时器 0
    TR0=1; //启动定时器 1
    ST=1;
    ST=0;
    while(1)
    {
        if(EOC==1)
        {
            OE=1;
            getdata=P0;
            OE=0;
            temp=getdata*235;
            temp=temp/128;
            i=5;
            dispbuf[0]=10;
            dispbuf[1]=10;
            dispbuf[2]=10;
            dispbuf[3]=10;
            dispbuf[4]=10;
```

```

        dispbuf[5]=0;
        dispbuf[6]=0;
        dispbuf[7]=0;
        while(temp/10)
        {
            dispbuf[i]=temp%10;
            temp=temp/10;
            i++;
        }
        dispbuf[i]=temp;
        ST=1;
        ST=0;
    }
}

```

//定时器 0 中断服务子程序

void t0(void) interrupt 1 using 0

```

{
    CLK=~CLK;
}

```

//定时器 1 中断服务子程序

void t1(void) interrupt 3 using 0

```

{
//重置 TH1
    TH1=(65536-4000)/256;
    TL1=(65536-4000)%256;
    P1=dispcode[dispbuf[dispcount]];
    P2=dispbitcode[dispcount];
    if(dispcount==7)
    {
        P1=P1 | 0x80;
    }
    dispcount++;
    if(dispcount==8)
    {
        dispcount=0;
    }
}
}

```

第 6 章 C51 单片机控制实例

单片机的特点在于控制应用，本章将主要介绍 C51 单片机的一些典型的控制应用实例。

6.1 基于ISD4004 芯片的语音录放设计

由单片机控制循环录放的语音电路，可作为录音机、复读机、音频记录仪使用，既可节省存储空间，又可降低成本，具有较高的实用价值。目前，市场上的固体录音机以及各种录音笔，大多采用的是顺序录音，不具备循环录音功能，一旦存储器录满，必须重新操作才行。

本例设计一种能够循环录放的语音电路，可解决上述问题。

6.1.1 实例说明

使用 ISD4004 语音录放芯片可实现语音的录放，该电路采用 AT89C51 单片机，通过操作按钮开关实现功能转换，操作命令由串行通信接口（SPI）送入。电路既可工作在顺序模式，又可工作在循环模式。当工作在循环模式的录音状态时，ISD4004 芯片将始终记录最后 16min 的语音信息，直至按下停止键。

6.1.2 ISD4004 介绍

ISD4004 是美国 ISD 公司出品的优质单片语音录放芯片，采用了专利技术——直接模拟存储技术（DAST TM），信号无需经过复杂的数字信号处理过程（如 D/A 转换、A/D 转换、数字压缩和语音合成），这样可以减少失真，而且音质好，适用于移动电话和其他便携式电子产品。ISD4004 内含 2 840 KB 大容量的闪速存储器，就能实现长达 16 min 的录音或放音；外围电路简单，体积小；3 V 单电源供电，耗电省，维持电流仅 1 μ A，可以和微控制器或微总线接口。由于取样频率 8.0 kHz、6.4 kHz、5.3 kHz、4.0 kHz 不同，相应的录放时间有 8 min、10 min、12 min、16 min 可供选择，频率越低，录放时间越长，但音质则有所下降。封装包括常见的 PDIP、SOIC、TSOP，还包括微型封装 CSP。芯片的工作温度在-40~+84℃之间。

芯片采用 CMOS 技术，内含振荡器、防混淆滤波器、平滑滤波器、音频放大器、自动静噪和高密度多电平闪烁存储阵列。芯片设计所有操作必须由 MCU（微控制器）控制，操作命令通过串行通信接口（SPI 或 Microwire）送入。片内信息存于闪烁存储器中，可在断电情况下保存 100 年（典型值），反复录音 10 万次。

1. 芯片引脚说明

ISD4004 的引脚如图 6-1 所示。

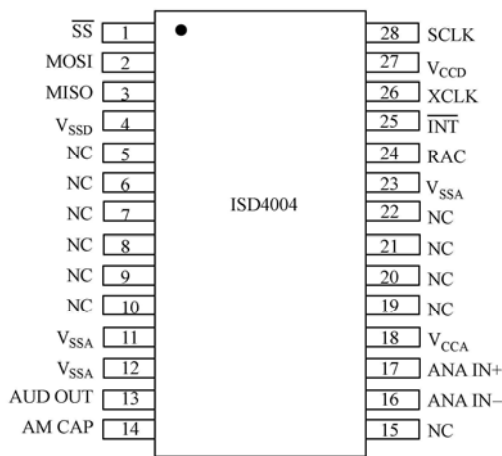


图 6-1 ISD4004 引脚图

ISD4004 的引脚功能说明如表 6-1 所示。

表 6-1 引脚说明

引脚名称	说 明
V_{CCA}, V_{CCD}	芯片的模拟电源和数字电源。芯片的模拟和数字电路应尽量使用不同的电源总线，并且分别引到外封装的不同引脚上，模拟和数字电源端最好分别走线，尽可能在靠近供电端处相连，而去耦电容应尽量靠近器件，这样可以降低噪声
V_{SSA}, V_{SSD}	芯片的模拟地和数字地。芯片内部的模拟和数字电路也使用不同的地线
ANA IN+	同相模拟输入。这是录音信号的同相输入端，输入放大器可用单端或差分驱动。单端输入时,信号由耦合电容输入，最大幅度为峰峰值 32 mV，耦合电容和本端的 3 k Ω 电阻的输入阻抗决定了芯片频带的低端截止频率。差分驱动时，信号最大幅度为峰峰值 16 mV
ANA IN-	反相模拟输入。差分驱动时，这是录音信号的反相输入端。信号通过耦合电容输入，最大幅度为峰峰值 16 mV
AUD OUT	音频输出。提供音频输出,可驱动 5 k Ω 的负载
\overline{SS}	片选。此端为低，即向该 ISD4004 芯片发送指令，两条指令之间为高电平
MOSI	串行输入。主控制器应在串行时钟上升沿之前半个周期将数据放到本端，供 ISD 输入
MISO	ISD 的串行输出端。ISD 未选中时，本引脚呈高阻态
SCLK	串行时钟。ISD 的时钟输入端,由主控制器产生，用于同步 MOSI 和 MISO 的数据传输。数据在 SCLK 上升沿锁存到 ISD,在下降沿移出 ISD
\overline{INT}	中断。本引脚为漏极开路输出，ISD 在任何操作（包括快进）中检测到 EOM 或 OVF 时，本引脚变低并保持，中断状态在下一个 SPI 周期开始时清除，中断状态也可用 RINT 指令读取
RAC	行地址时钟。漏极开路输出，每个 RAC 周期表示 ISD 存储器的操作进行了一行（ISD4004 系列中的存储器共 2 400 行）。该信号 175 ms 保持高电平，低电平为 25 ms。快进模式下，RAC 的 218.75 μ s 是高电平，31.25 μ s 为低电平。该引脚可用于存储管理技术
XCLK	外部时钟。本引脚内部有下拉元件，芯片内部的采样时钟在出厂前已调校，误差在 $\pm 1\%$ 内。商业级芯片在整个温度和电压范围内，频率变化在 $\pm 2.25\%$ 内。若要求更高的精度，可从本引脚输入外部时钟。由于内部的防混浊及平滑滤波器已设定，故上述推荐的时钟频率不应改变。输入时钟的占空比无关紧要，因而内部首先进行了分频。在不外接地时钟时此端必须接地
AM CAP	自动静噪。当录音信号电平下降到内部设定的某一阈值以下时，自动静噪功能使信号衰弱，这样有助于消除无信号（静音）时的噪声。通常本引脚对地接 1 μ F 的电容，构成内部信号电平峰值检测电路的一部分。检出的峰值电平与内部设定的阈值作比较，决定自动静噪功能的翻转点。大信号时，自动静噪电路不衰减，静音时衰减 6 dB。1 μ F 的电容也影响自动静噪电路对信号幅度的响应速度。本引脚接 V_{CCA} 则禁止自动静噪

2. 内部结构及时序图

ISD4004 的内部结构框图如图 6-2 所示。

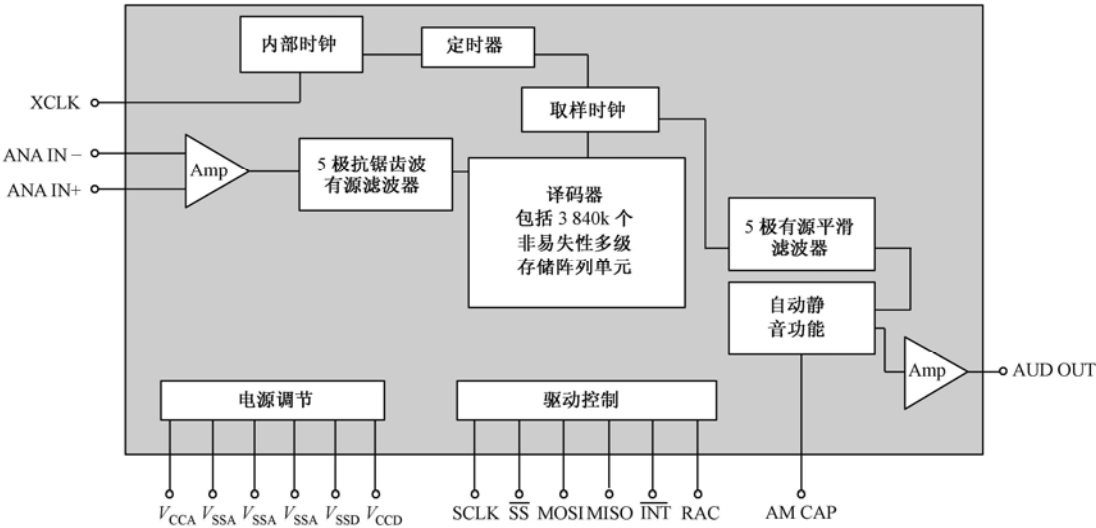


图 6-2 ISD4004 内部结构框图

ISD4004 的时序如图 6-3 所示。

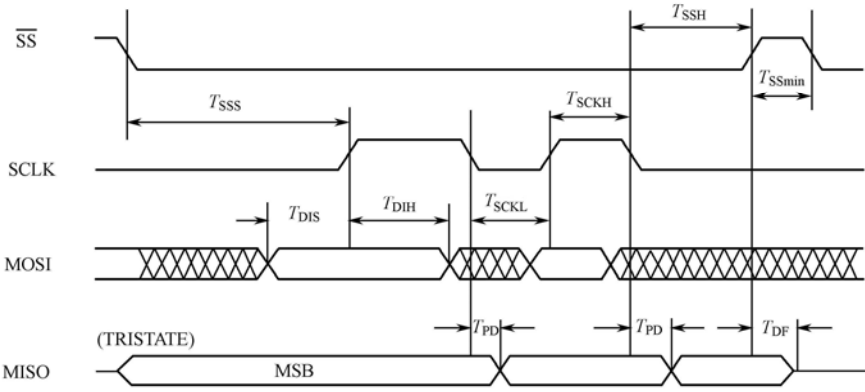


图 6-3 ISD4004 时序图

ISD4004 录播及停止时的时序如图 6-4 所示。

3. ISD4004 的三种典型应用

- ISD4004 的典型应用（1）——使用串行外设接口的典型应用，其原理如图 6-5 所示。
- ISD4004 的典型应用（2）——使用 Microwire 总线的典型应用，其原理如图 6-6 所示。
- ISD4004 的典型应用（3）——使用微处理器上的串行外设接口的典型应用，其原理如图 6-7 所示。

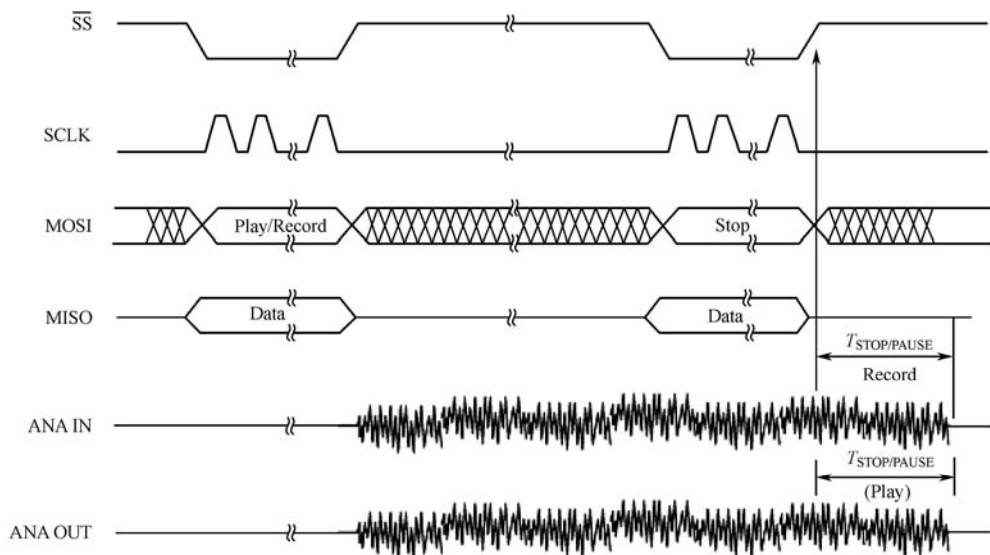


图 6-4 ISD4004 录播及停止时的时序图

4. 串行外设接口——SPI

ISD4004 工作于 SPI 串行接口模式。SPI 协议是一个同步串行数据传输协议，协议假设 MCU 的 SPI 移位寄存器在 SCLK 的下降沿动作，所以对于 ISD4004，在时钟上升沿锁存 MOSI 引脚的数据，在下降沿将数据送至 MISO 引脚。

协议的内容如下：

- 所有串行数据传输开始于 \overline{SS} 下降沿；
- \overline{SS} 在传输期间保持为低电平，在两条指令之间则保持为高电平；
- 数据在时钟上升沿移入，在下降沿移出；
- \overline{SS} 变低，当输入指令和地址后，ISD4004 才能开始录放操作；
- 指令格式是 8 位控制码加 16 位地址码；
- ISD4004 的任何操作（含快进）如遇到 EOM 或 OVF，则产生一个中断，在下一个 SPI 周期开始时清除该中断；
- 使用读指令使中断状态位移出 ISD4004 的 MISO 引脚时，控制及地址数据也同步从 MOSI 端移入。但要注意的是，移入的数据是否与器件当前进行的操作兼容。同样也允许在一个 SPI 周期里，同时执行读状态和开始新的操作。所有操作在运行位（RUN）置 1 时开始，置 0 时结束；
- 所有指令都在 \overline{SS} 端上升沿开始执行。

SPI 端口的控制位如图 6-8 所示。

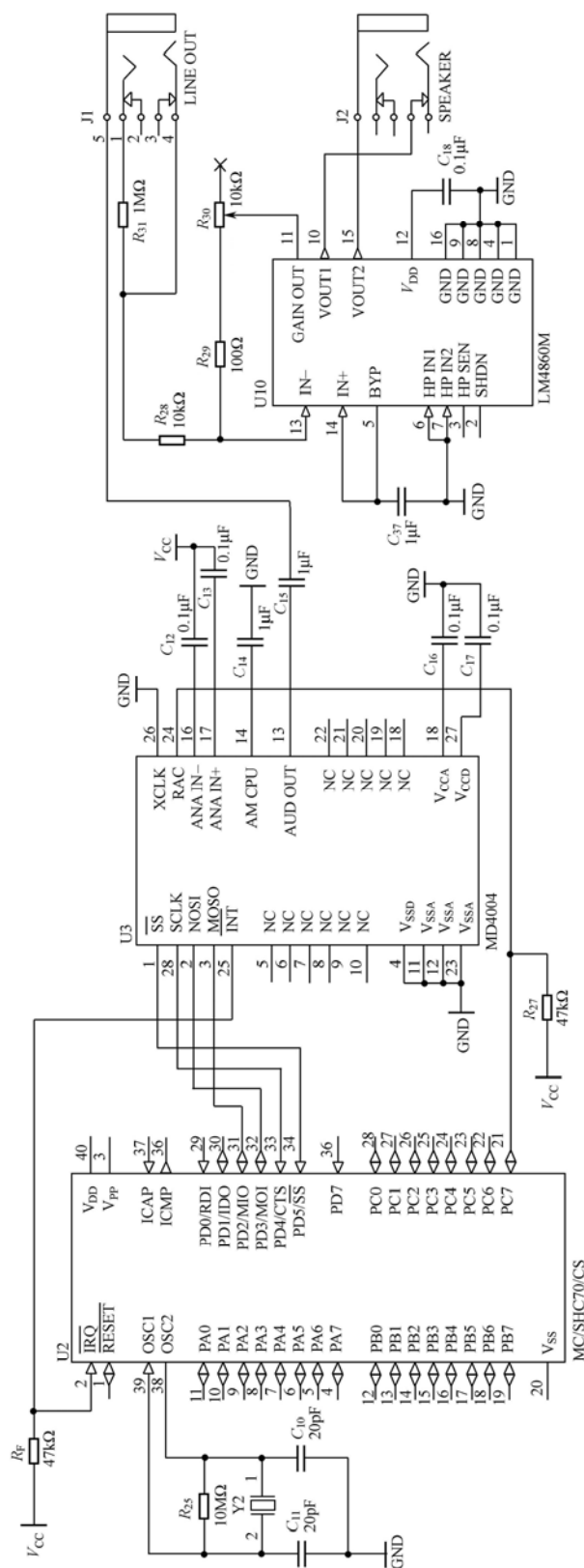


图 6-5 使用串行外设接口的典型应用

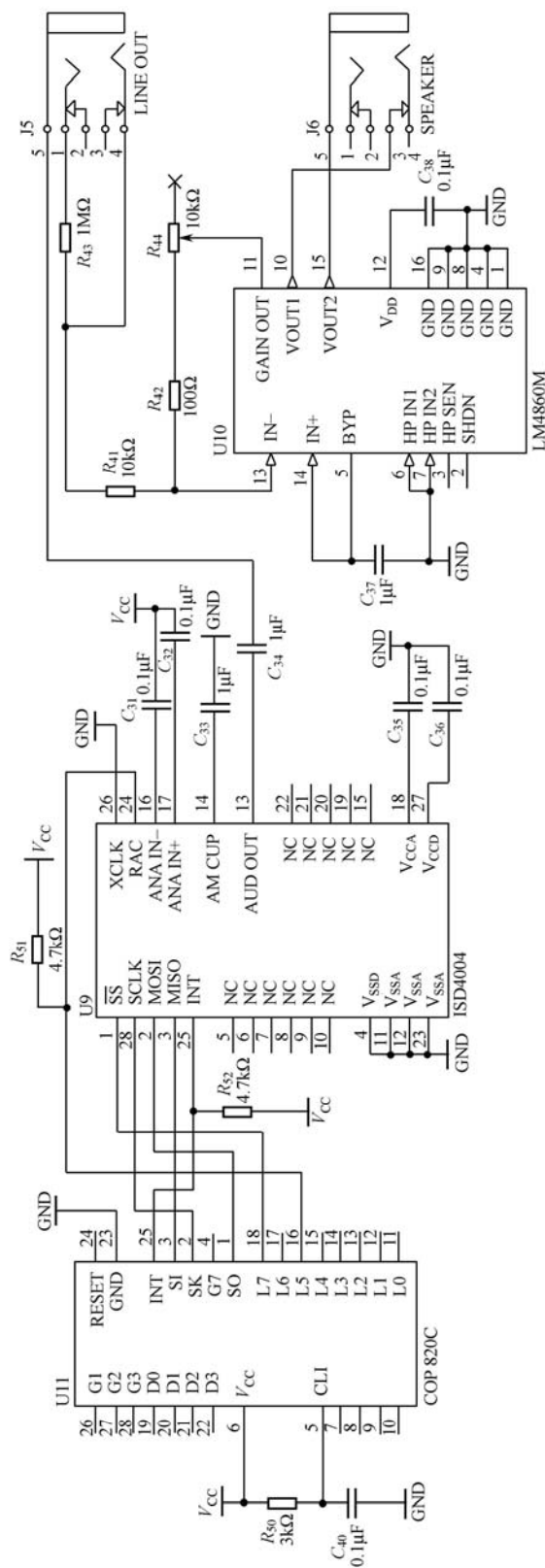


图 6-6 使用 Microwire 总线的典型应用

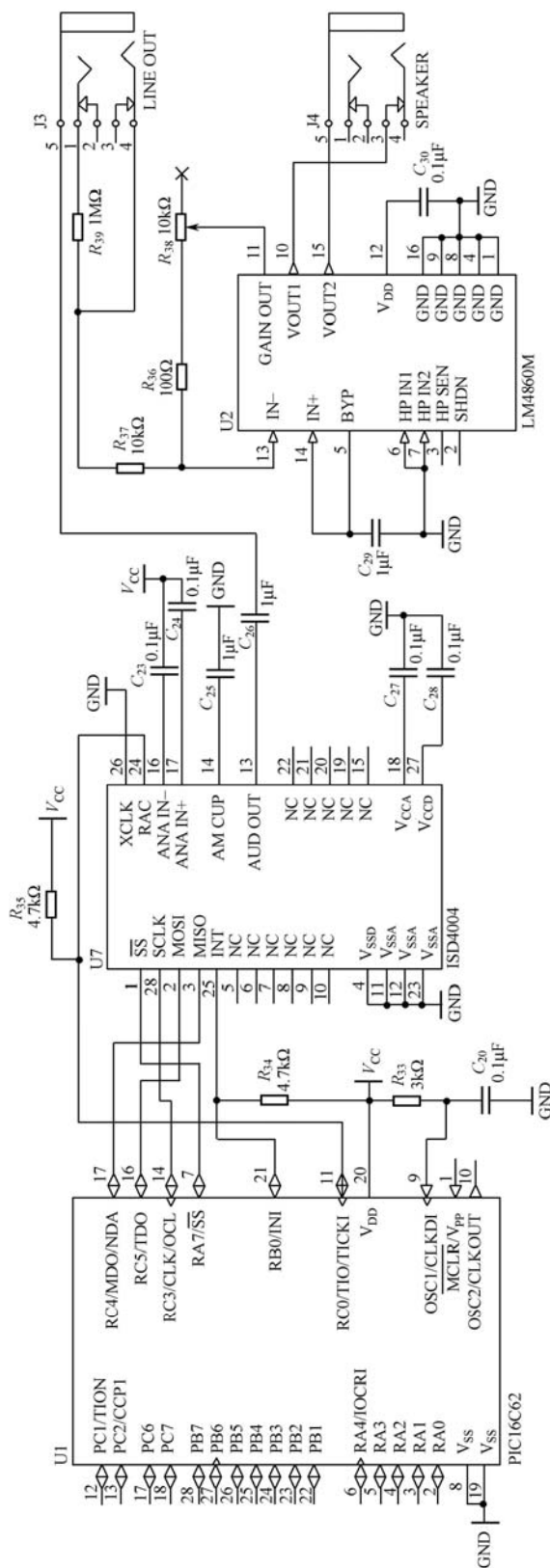


图 6-7 使用微处理器上的串行外接口的典型应用

表 6-3 ISD4004 指令表

指 令	8 位控制码<16 位地址>	操 作
POWER UP	00100XXX<XXXXXXXXXXXXXXXXXX>	上电：等待 TPUD 后器件可以工作
SET PLAY	11100XXX<A15~A0>	从指定地址开始放音，后跟 PLAY 指令可使放音继续进行下去
PLAY	11110XXX<XXXXXXXXXXXXXXXXXX>	从当前地址开始放音（直至 EOM 或 OVF）
SET REC	10100XXX<A15~A0>	从指定地址开始录音，后跟 REC 指令可使录音继续进行下去
REC	10110XXX<XXXXXXXXXXXXXXXXXX>	从当前地址开始录音（直至 OVF 或停止）
SET MC	11101XXX<A15~A0>	从指定地址开始快进，后跟 MC 指令可使快进继续进行下去
MC	11111XXX<XXXXXXXXXXXXXXXXXX>	执行快进，直到 EOM。若再无信息，则进入 OVF 状态
STOP	0X110XXX<XXXXXXXXXXXXXXXXXX>	停止当前操作
STOP WRDN	0X01XXX<XXXXXXXXXXXXXXXXXX>	停止当前操作并掉电
RINT	0X110XXX<XXXXXXXXXXXXXXXXXX>	读状态：OVF 和 EOM

6.1.3 硬件设计

ISD4004 与 C51 单片机的硬件连接如图 6-10 所示，P1.0 接 SS，P1.1 接 SCLK，P1.2 接 MOSI，P1.3 接 MISO，P3.4 接 IN 口。P1.4、P1.5、P1.6、P1.7 接外围电路。

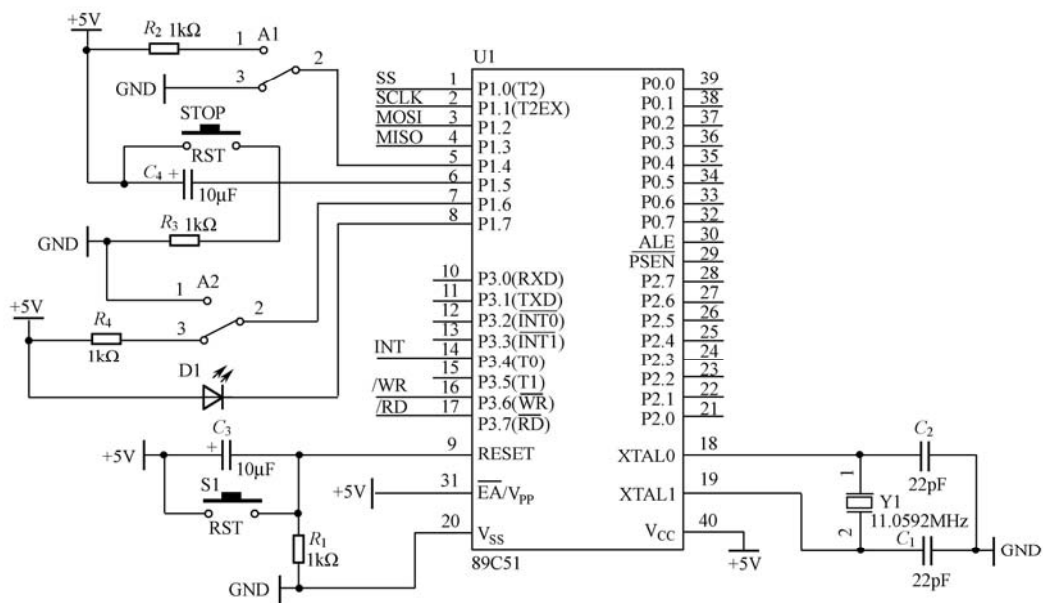
当开关 A1 扳向上面时，是录音状态，开关 A2 扳向下面时（相当于按住按键），指示灯亮，即可对着话筒讲话录音，松开按键时录音停止并生成一段录音。同样操作，则开始录制下一段。按下 STOP 键为复位，再录音时又从第一段开始。当开关 A2 扳向下面时，是放音状态，开关 A2 扳向下面时，即播放一段录音，该录音结束后自动停止放音。按下 STOP 键复位，放音时从第一段开始。

6.1.4 软件设计

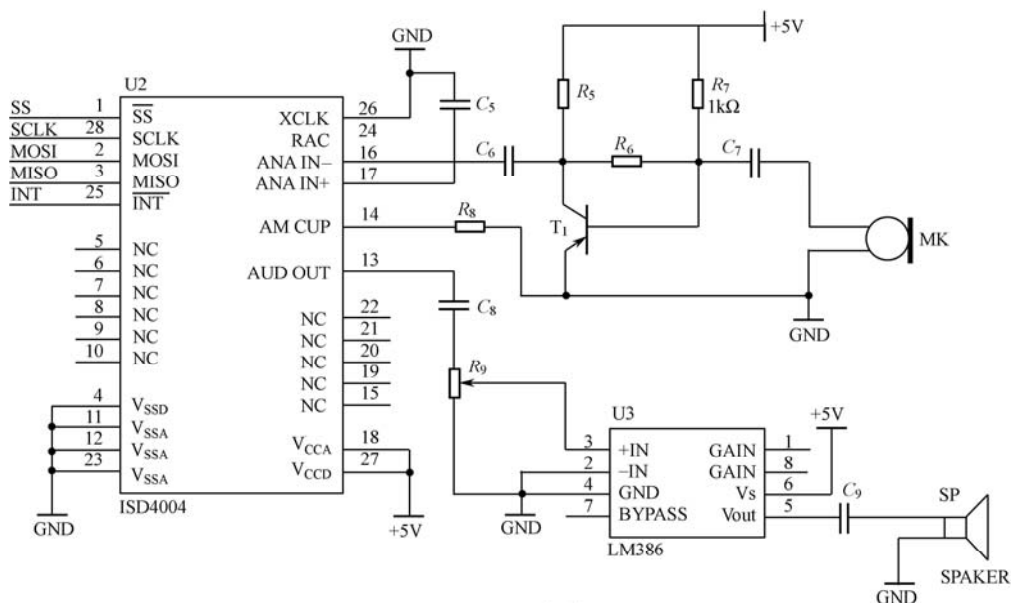
软件的设计要结合硬件连线等具体情况。AT89C51 单片机提供了用户键盘和 ISD4004 所需接口。它接收按键动作，并将相应指令传给 ISD4004，同时监控 ISD4004 的中断输出。当开关闭合时，读取 ISD4004 的状态寄存器，从而根据 OVF 和 EOM 的状态进行相应的处理。当 OVF=1，即存储器溢出时，则不管当前为何种状态均将 ISD4004 的地址置零，并继续运行原指令；当 EOM=1 时，当前状态只可能为放音或快进，若为快进则置为放音态，并继续运行。如此设计便可实现循环录放的功能，同时在快进时，自动停止在下一个语音段开始处，并继续放音。

1. 程序流程图

程序流程如图 6-11 所示。



(a) 单片机控制部分



(b) ISD4004 部分

图 6-10 ISD4004 与 C51 单片机的硬件连接图

2. 程序代码

本例的程序代码说明如下：

```
/*AT89C51 12MHz*/
#include <reg51.h>
sbit PX=P1^0; //片选
```

```

sbit   CLK=P1^1;           //ISD4004 时钟
sbit   DIN=P1^2;           //数据输入
sbit   DOUT=P1^3;          //数据输出
sbit   LED=P1^7;           //指示灯
sbit   AN=P1^6;            //执行
sbit   RESET=P1^5;         //复位
sbit   PR=P1^4;            //PR=1 录音, PR=0 放音
sbit   Interrupt=P3^4;     //中断

```

```

void   delay50();
void   FUN(unsigned char d);
void   START();
void   PLAUSE();           //停止当前操作
void   DROP();

```

```

main()
{
    unsigned char delay;    //延时参数
    unsigned char isdlow,isdhigh; //ISD 高低位
    unsigned char n1;

```

```

    SP=0x10;
    P0=0xFF;
    P1=0xFF;
    P2=0xFF;
    P3=0xFF;
    EA=0;

11:
    LED=1;                //关指示灯
    DROP();               //ISD 掉电
    delay=200;
    while(AN);            //等按 AN 键
    while(delay--);       //延时
    START();              //ISD 上电
    isdlow=isdhigh=0;     //ISD 高低位地址置 0
    if(PR)                //PR=1 录音
    {
        FUN(isdlow);
        FUN(isdlow);
        FUN(isdhigh);
        PX=1;

```

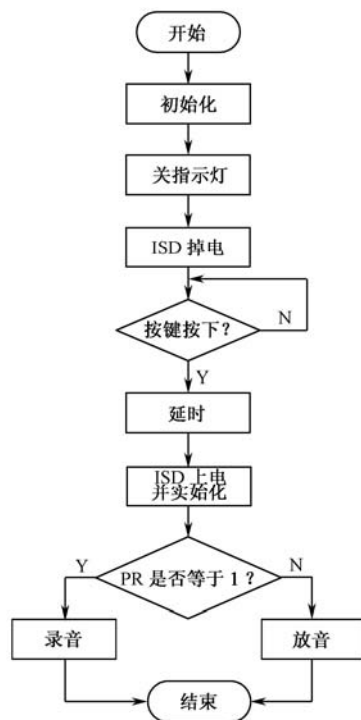


图 6-11 程序流程图

```

l2:
    n1=10;
    while(n1—)
        delay50();
LED=0;
FUN(0xB0);
PX=1;
if(Interrupt)
{
    while(!AN);
    n1=200;
    while(n1—);
    LED=1;                //关指示灯
    PLAUSE();             //停止当前操作
    if(RESET)
    {
        while(!AN);
        PLAUSE();
        goto l2;
    }
}
else
{
    l3:
    LED=1;
    n1=15;
    while(!AN)
    {
        while(!n1—)
            delay50();
        LED=0;
        n1=15;
        while(!AN)
        {
            while(!n1—)
                delay50();
            goto l3;
        }
    }
}
}
}
}

```



```
else
```

```
//PR=0 放音
```

```
{
```

```
while(!AN);
```

```
FUN(isdlow);
```

```
FUN((isdhigh|0xE0)&0xE7);
```

```
PX=1;
```

```
l4:
```

```
LED=0;
```

```
FUN(0xF0);
```

```
PX=1;
```

```
if(RESET)
```

```
{
```

```
while(Interrupt);
```

```
LED=1;
```

```
PLAUSE();
```

```
PX=0;
```

```
CLK=0;
```

```
CLK=1;
```

```
if(!DOUT)
```

```
{
```

```
CLK=0;
```

```
PX=1;
```

```
PLAUSE();
```

```
if(RESET)
```

```
{
```

```
while(!AN);
```

```
goto l4;
```

```
}
```

```
}
```

```
}
```

```
}
```

```
CLK=0;
```

```
PX=1;
```

```
PLAUSE();
```

```
goto l1;
```

```
}
```

```
//ISD4004 芯片驱动，延时 50 ms
```

```
void delay50()
```

```
{
```

```
TMOD=1;
```

```

TH0=0x3C;                                //50 ms 延时初值置入
TL0=0xB0;
TR0=1;
while(!TF0);
TF0=0;
TR0=0;
}
//ISD3300.4003 SPI 写入程序
void FUN(unsigned char d)
{
    unsigned char i,j;
    PX=0;                                //开片选
    CLK=0;                                //时钟 CLK=0
    j=d;
    for(i=0;i<8;i++)
    {
        if(j&0x01)                        //数据写 DIN
            DIN=1;
        else
            DIN=0;
        CLK=1;                            //时钟 CLK=1
        j=j>>1;
        CLK=0;                            //时钟 CLK=0
    }
}

/*ISD 上电*/
void START()
{
    FUN(20);                              //发 00100XXXXXXXXXXXXX
    PX=1;                                //关片选
    delay50();                            //50 ms 延时
    delay50();                            //50 ms 延时
}

/*停止当前操作*/
void PLAUSE()
{
    FUN(30);                              //发 0X110XXX
    PX=1;                                //关片选
    delay50();                            //50 ms 延时
}

```

```

delay50();                                //50 ms 延时
}

//停止当前操作掉电
void DROP()
{
FUN(10);                                //发 0X010XXXXXXXXXX
PX=1;                                    //关片选
delay50();                                //50 ms 延时
delay50();                                //50 ms 延时

```

6.2 单片机密码锁设计

随着科技的发展，安全已成为人们关注的焦点之一，于是各种安全产品相继问世，比如指纹防盗、红外防盗和 GPS 等，虽然这类产品安全性高，但因其生产成本低，携带、安装、使用不方便，在一定程度上限制了这类产品的普及和推广。而电子密码在日常生活和工作中却得到广泛应用，比如智能提款机、公司的转账交易、小区单元的电子密码锁。

本例将主要介绍一种基于 C51 单片机智能锁的硬件和软件设计及实现方法。

6.2.1 实例说明

本例是基于单片机实现电子密码锁，这种电路设计具有按键有效提示、输入错误提示，控制开锁电平、控制报警电路、修改密码等多种功能。一个简单的密码锁就是一个小型的单片机系统，它应该具有输入/输出设备，利用键盘可以输入密码，还可以实现密码的确认、取消和修改；利用数码管或者显示屏查看自己输入的密码正确与否，利用蜂鸣器实现警告提示。

6.2.2 设计思路分析

电子密码锁内部的单片机是核心处理设备，它负责获取由键盘输入的密码，将输入的密码和预设的密码进行比较，如果相同则产生相应的输出；如果输入的密码与预设值不同，则提示重新输入，并且记录下用户输入错误密码的次数，若输入的错误次数超过预设的限制次数，则采取相应的保护措施，防止他人反复试探密码。

在产品的设计过程中，实现人机交互越来越重要，所以键盘与显示是必不可少的配置，接下来介绍一下键盘接口电路。

1. 键盘接口介绍

按键是一种常开型按钮开关，不使用时两个触点处于断开状态，按下时它们才闭合。键盘是一组按键的组合，它是最常用的单片机输入设备，操作人员可以通过按键来输入数据或符号，实现简单的人机通信。键盘可以分为独立连接式和矩阵式两类。

独立连接式键盘的每个键都独立的接入一根数据线，是最简单的键盘电路，所有的数

据输入线都被连接成高电平，当有按键按下时，与之相连的数据输入线被拉成低电平。这种按键的优点是结构简单，使用方便，但是占用的 I/O 口比较多。

矩阵式键盘适用于按键较多的场合，按键位于行列的交叉点上，行列线分别接到按键的两端，列线通过上拉电阻接到+5V，无按键动作时，列线处于高电平状态，有按键按下时，列线电平将由与此列线相连的行线电平决定。如果行线为低电平，则列线为低电平，行线为高电平，则列线为高电平。这一点是识别按键是否被按下的关键所在。矩阵式键盘要比独立连接式键盘节约很多的 I/O 口。一个 3×3 的行列结构可以构成由 9 个按键的键盘。同理，一个 4×4 的行列结构可以构成 16 个按键的键盘。本例中的 4×4 的矩阵式键盘如图 6-12 所示。

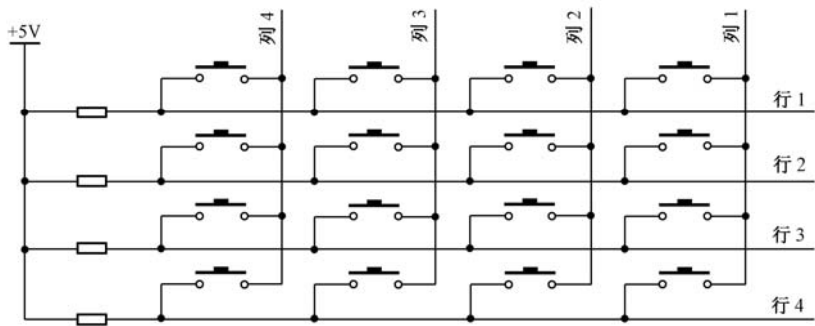


图 6-12 4×4 的矩阵式键盘示意图

本例中电子密码锁对应的 4×4 矩阵式按键的功能分布如图 6-13 所示。

通过行列键盘扫描的方法可以获取键盘输入的键值。

下面介绍一下 4×4 的矩阵式键盘的识别过程。

① 假定这 16 个按键各代表十六进制的 0~F，键被按下时，与此线相连的列线电平，将由与此线相连的行线电平决定。而列线电平在无键按下时处于高电平，如果让所有的行线处于高电平，那么无论按键按下与否，都不会引起列电平变化的，始终是高电平。所以，就必须让所有的行线处于低电平，当有按键按下时，按键所在的列线就被拉成低电平。

② 这时还不能确定到底该列中哪一只按键被按下，因为同一列上任何一只按键按下，均会产生同样的效果，所以，让所有行线处于低电平只能得出某列有按键被按下的结论。为了进一步判断到底哪一行的按键被按下，可以在某一时刻只让一条行线处于低电平，其余所有行线处于高电平。

③ 当第一行为低电平，其余各行为高电平时，假定第一行与第一列交叉点的按键被按下第一列为低电平，其余各列为高电平。这样就可以确定是第一行和第一列交叉点处的按键被按下。

④ 要识别一个按键，最少从单片机的 I/O 口低 4 位输出一次数，据从 I/O 口高 4 位读回一次数据，这称为一次扫描。当按下第二行的按键时，需要 2 遍扫描；第三行需要 3 遍扫描；第四行需要 4 遍扫描。

由于机械触点的弹性振动，按键在按下时不会马上稳定地接通，而在弹起时也不能马上完全断开，因而在按键闭合和断开的瞬间均会出现一连串的抖动，这称为按键的抖动干扰，其产生的波形如图 6-14 所示。

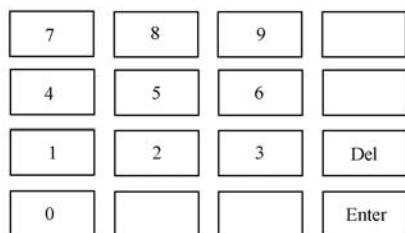


图 6-13 电子密码锁按键序号排列

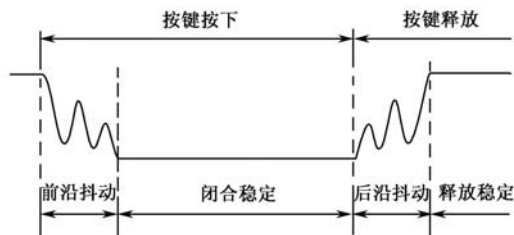


图 6-14 抖动干扰产生的波形

当按键按下时会产生前沿抖动，当按键弹起时会产生后沿抖动，这是所有机械触点式按键在状态输出时的共性问题。抖动的时间长短取决于按键的机械特性与操作状态，通常为 $10 \sim 100 \text{ ms}$ 。

按键的抖动会造成按一次按键产生的开关状态被 CPU 误读几次，为了使 CPU 能正确地读取按键的状态，必须在按键闭合或断开时，消除产生的前沿抖动或后沿抖动，消除抖动的方法有硬件方法和软件方法两种。

(1) 硬件方法

硬件方法是设计一个滤波延时电路或单稳态电路等硬件电路来避开按键的抖动时间。

图 6-15 是由 R_2 和 C 组成的滤波延时消抖电路，设置在按键 S 与 CPU 数据线 D_i 之间。

按键 S 未被按下时，电容两端电压为 0，即与非门输入 V_i 为 0，输出 V_o 为 1。当 S 按下时，由于 C 两端电压不能突变，充电电压 V_i 在充电时间内未达到与非门的开启电压，门的输出 V_o 将不会改变，直到充电电压 V_i 大于门的开启电压时，与非门的输出 V_o 才变为 0，这段充电延迟时间取决于 R_1 、 R_2 和 C 值的大小，电路设计时只要使之大于或等于 100 ms 即可消除按键抖动的影响。同理，按键 S 断开时，即使出现抖动，由于 C 的放电延迟过程，也会消除按键抖动的影响。

如图 6-16 所示， $V1$ 是未施加滤波电路含有前沿抖动、后沿抖动的波形， $V2$ 是施加滤波电路后消除抖动的波形。

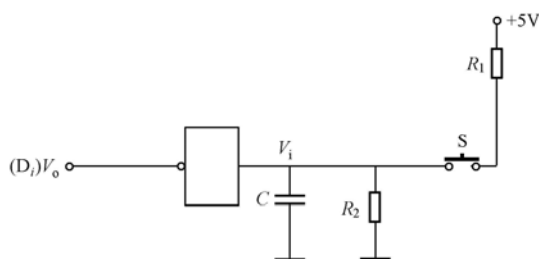


图 6-15 滤波延时消抖电路

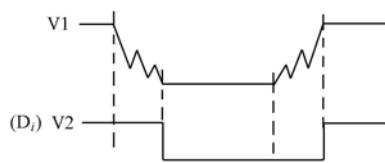


图 6-16 滤波电路抖动波形

(2) 软件方法

软件方法是指编写一段时间大于 100 ms 的延时程序，在第一次检测到有按键按下时，执行这段延时子程序使按键的前沿抖动消失后再检测该按键的状态，如果该按键仍保持闭合状态电平，则确认为该按键已被稳定地按下，否则无按键按下，从而消除了按键抖动的影响。同理，在检测到按键释放后，也同样要延迟一段时间，以消除后沿抖动，然后再转入对该按键的处理。

通常情况下，我们不对按键释放的后沿抖动进行处理。实际应用中，对按键的要求也是千差万别的，要根据不同的需要来编制处理程序，但以上是消除键抖动的原则。

2. 数码管显示原理

在单片机中，通常使用由七段 LED 发光二极管和一个点状发光二极管构成的数码管进行显示。数码管的内部结构示意图，如图 6-17 所示。

当在发光二极管某字段施加一定的正电压时，该字段所对应的笔画发光，不加正电压则不发光。为了保护各段 LED 不被损坏，需外加限流电阻，不管是共阴极还是共阳极，它们的 8 段排列顺序都是一样的，如图 6-18 所示。这 8 段的名称分别为 a、b、c、d、e、f、g 段和 dp 段（显示小数点）。

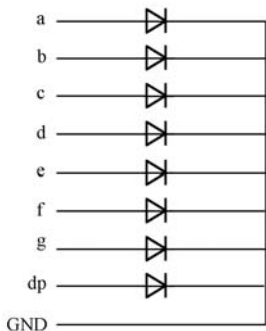


图 6-17 数码管的内部结构示意图

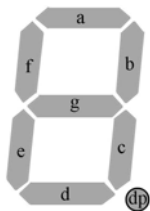


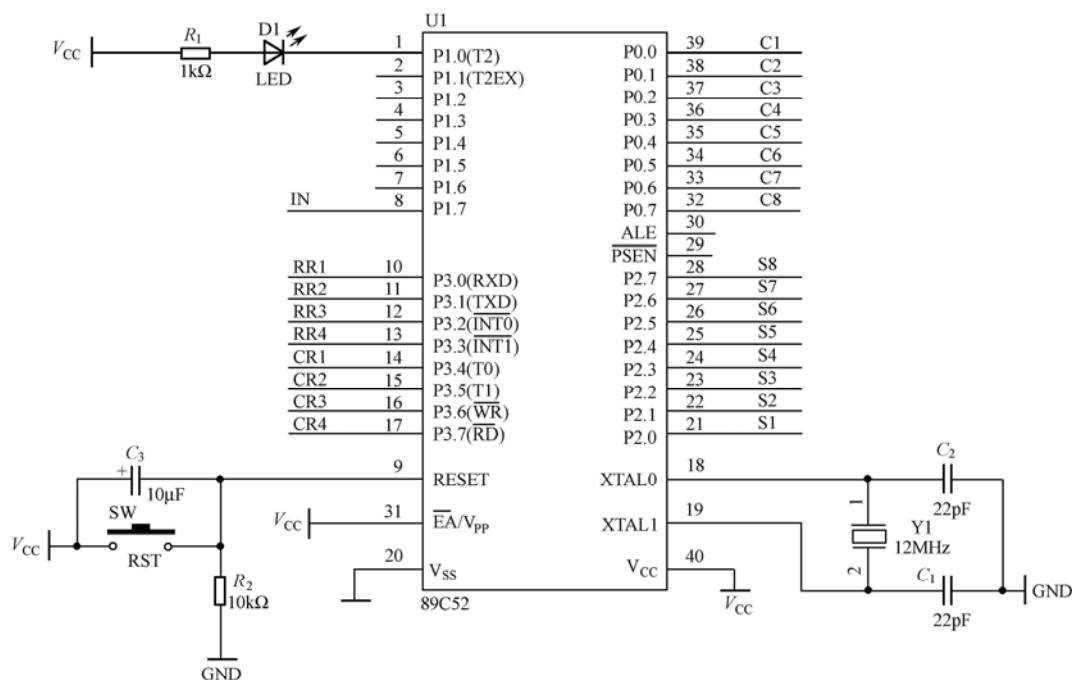
图 6-18 LED 排列顺序

6.2.3 硬件电路设计

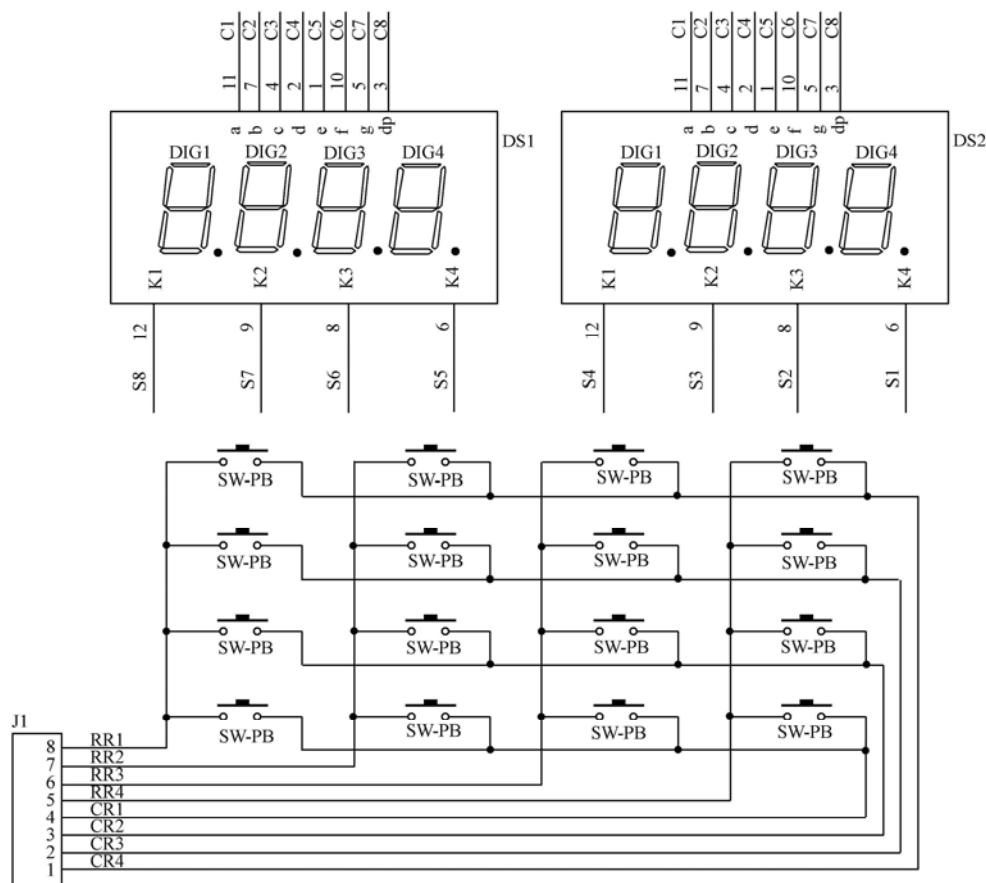
本实例简单介绍用 4×4 的矩阵式键盘组成 0~9 数字键、确认键以及取消键；用 8 位七段数码管组成显示电路提示信息，当输入密码时，只显示“8”，当密码位数输入完毕按下确认键时，对输入的密码与设定的密码进行比较，若输入的密码正确，发出“叮咚”声，同时门被打开；若密码不正确，禁止按键输入 3 s，同时发出“滴滴”的报警声；若在 3 s 之内仍有按键按下，则禁止按键输入 3 s 被重新禁止。

4×4 矩阵式键盘及 8 位数码管显示构成的电子密码锁的电路原理如图 6-19 所示。

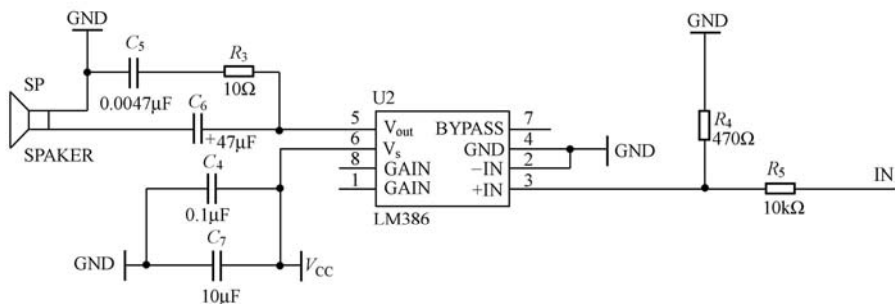
单片机中的 P0.0~P0.7 口用 8 芯排线连接到动态数码显示区域中的 a、b、c、d、e、f、g、dp 脚上；单片机中的 P2.0~P2.7 口用 8 芯排线连接到动态数码显示区域中的 S1、S2、S3、S4、S5、S6、S7、S8 脚上；单片机中的 P3.0~P3.7 口用 8 芯排线连接到 4×4 行列式键盘区域中的 RR1、RR2、RR3、RR4、CR1、CR2、CR3、CR4 脚上；单片机中的 P1.0 用导线连接到 8 路发光二极管模块区域中的 L2 引脚上；单片机中的 P1.7 口用导线连接到音频放大模块区域中的 IN 引脚上；音频放大模块区域中的 OUT 接到喇叭上。



(a) 单片机部分



(b) 密码锁键盘、数码管部分



(c) 音频发生器部分

图 6-19 电子密码锁的电路原理图

6.2.4 软件设计

8 位数码显示初始化时, 显示 P, 接着输入最大 6 位数的密码, 当密码输入完后, 按下确认键, 进行密码比较, 然后给出相应的信息。在输入密码过程中, 显示器只显示 8。当输入的数字超过 6 个时, 给出报警信号。在密码输入过程中, 若输入错误, 可以利用 DEL 按键删除刚才输入的错误的数字。

1. 设计流程

4×4 矩阵式键盘及 8 位数码管显示构成的电子密码锁的软件设计流程如图 6-20 所示。

2. 程序说明

4×4 键盘及 8 位数码管显示构成的电子密码锁的程序说明如下。

```
#include <at89x52.h>
#define uchar unsigned char
#define uint unsigned int
uchar ps[]={1,2,3,4,5};
uchar code dispbit[]={0xfe,0xfd,0xfb,0xf7,
                       0xef,0xdf,0xbf,0x7f}; //显示位
uchar code dispcode[]={0x3f,0x06,0x5b,0x4f,0x66,
                       0x6d,0x7d,0x07,0x7f,0x6f,
                       0x77,0x7c,0x39,0x5e,0x79,0x71,
                       0x00,0x40,0x73,0xff};
uchar dispbuf[8]={18,16,16,16,16,16,16,16};
uchar dispcount; //显示计数器
uchar flashcount;
uchar temp;
```

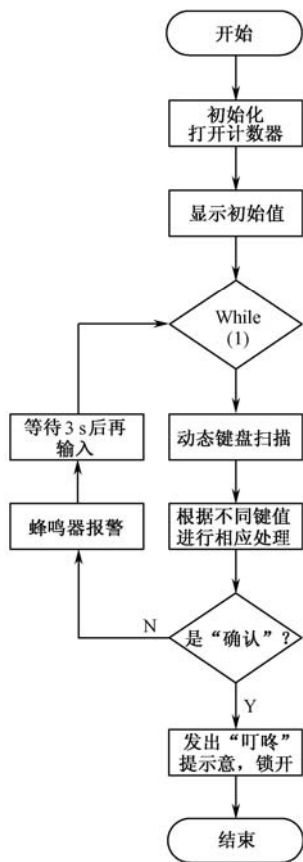


图 6-20 软件设计流程图

//显示码
//显示缓冲


```

uchar key;
uchar keycount;           //按键计数器
uchar pslen=5;
uchar getps[6];
bit keyoverflow;          //键值溢出标志位
bit errorflag;            //错误标志位
bit rightflag;            //正确标志位
uint second3;
uint aa,bb;
uint cc;
bit okflag;               //OK 标志位
bit alarmflag;            //报警标志位
bit hibitflag;
uchar oka,okb;
void main(void)
{
    uchar i,j;
    TMOD=0x01;  // t0 工作方式 1
    //重置定时
    TH0=(65536-500)/256;
    TL0=(65536-500)%6;
    TR0=1;        //启动计数器 0
    ET0=1;
    EA=1;
    while(1)
    {
        P3=0xff;
        P3_4=0;
        temp=P3;
        temp=temp & 0x0f;
        if (temp!=0x0f)
        {
            for(i=10;i>0;i--)
            for(j=248;j>0;j--);
            temp=P3;
            temp=temp & 0x0f;
            if (temp!=0x0f)
            {
                temp=P3;
                temp=temp & 0x0f;
                switch(temp)

```

```

{
    case 0x0e:
        key=7;
        break;
    case 0x0d:
        key=8;
        break;
    case 0x0b:
        key=9;
        break;
    case 0x07:
        key=10;
        break;
}
temp=P3;
P1_1=~P1_1;
if((key>=0) && (key<10))
{
    if(keycount<6)
    {
        getps[keycount]=key;
        dispbuf[keycount+2]=19;
    }
    keycount++;
    if(keycount==6)
    {
        keycount=6;
    }
    else if(keycount>6)
    {
        keycount=6;
        keyoverflag=1;           //键值溢出
    }
}
else if(key==12)                //删除键值
{
    if(keycount>0)
    {
        keycount--;
        getps[keycount]=0;
        dispbuf[keycount+2]=16;
    }
}

```

```

        }
        else
        {
            keyoverflag=1;
        }
    }
    else if(key==15)        //进入键值
    {
        if(keycount!=pslen)
        {
            errorflag=1;
            rightflag=0;
            second3=0;
        }
        else
        {
            for(i=0;i<10;i++)
            {
                if(getps[i]!=ps[i])
                {
                    i=keycount;
                    errorflag=1;
                    rightflag=0;
                    second3=0;
                    goto a;
                }
            }
            errorflag=0;
            rightflag=1;
            i=keycount;
a:        }
        }
    }
    temp=temp & 0x0f;
    while(temp!=0x0f)
    {
        temp=P3;
        temp=temp & 0x0f;
    }
    keyoverflag=0;
}
}

```

```

P3=0xff;
P3_5=0;
temp=P3;
temp=temp & 0x0f;
if (temp!=0x0f)
{
    for(i=10;i>0;i--)
    for(j=248;j>0;j--);
    temp=P3;
    temp=temp & 0x0f;
    if (temp!=0x0f)
    {
        temp=P3;
        temp=temp & 0x0f;
        switch(temp)
        {
            case 0x0e:
                key=4;
                break;
            case 0x0d:
                key=5;
                break;
            case 0x0b:
                key=6;
                break;
            case 0x07:
                key=11;
                break;
        }
        temp=P3;
        P1_1=~P1_1;
        if((key>=0) && (key<10))
        {
            if(keycount<6)
            {
                getps[keycount]=key;
                dispbuf[keycount+2]=19;
            }
            keycount++;
            if(keycount==6)
            {

```

```

        keycount=6;
    }
    else if(keycount>6)
    {
        keycount=6;
        keyoverflag=1;           //键值溢出
    }
}
else if(key==12)                //删除键值
{
    if(keycount>0)
    {
        keycount--;
        getps[keycount]=0;
        dispbuf[keycount+2]=16;
    }
    else
    {
        keyoverflag=1;
    }
}
else if(key==15)                //进入键值
{
    if(keycount!=pslen)
    {
        errorflag=1;
        rightflag=0;
        second3=0;
    }
    else
    {
        for(i=0;i<10;i++)
        {
            if(getps[i]!=ps[i])
            {
                i=keycount;
                errorflag=1;
                rightflag=0;
                second3=0;
                goto a4;
            }

```

```

        }
        errorflag=0;
        rightflag=1;
a4:        i=keycount;
        }

    }

    temp=temp & 0x0f;
    while(temp!=0x0f)
    {
        temp=P3;
        temp=temp & 0x0f;
    }
    keyoverflowflag=0;
}
}

```

```

P3=0xff;
P3_6=0;
temp=P3;
temp=temp & 0x0f;
if (temp!=0x0f)
{
    for(i=10;i>0;i--)
    for(j=248;j>0;j--);
    temp=P3;
    temp=temp & 0x0f;
    if (temp!=0x0f)
    {
        temp=P3;
        temp=temp & 0x0f;
        switch(temp)
        {
            case 0x0e:
                key=1;
                break;
            case 0x0d:
                key=2;
                break;
            case 0x0b:
                key=3;
                break;

```

```

        case 0x07:
            key=12;
            break;
    }
temp=P3;
P1_1=~P1_1;
if((key>=0) && (key<10))
{
    if(keycount<6)
    {
        getps[keycount]=key;
        dispbuf[keycount+2]=19;
    }
    keycount++;
    if(keycount==6)
    {
        keycount=6;
    }
    else if(keycount>6)
    {
        keycount=6;
        keyoverflag=1;           //键值溢出
    }
}
else if(key==12)                //删除键值
{
    if(keycount>0)
    {
        keycount--;
        getps[keycount]=0;
        dispbuf[keycount+2]=16;
    }
    else
    {
        keyoverflag=1;
    }
}
else if(key==15)//enter key
{
    if(keycount!=pslen)
    {

```

```

        errorflag=1;
        rightflag=0;
        second3=0;
    }
    else
    {
        for(i=0;i<10;i++)
        {
            if(getps[i]!=ps[i])
            {
                i=keycount;
                errorflag=1;
                rightflag=0;
                second3=0;
                goto a3;
            }
        }
        errorflag=0;
        rightflag=1;
        i=keycount;
a3:    }
    }

    temp=temp & 0x0f;
    while(temp!=0x0f)
    {
        temp=P3;
        temp=temp & 0x0f;
    }
    keyoverflag=0;
}

}

P3=0xff;
P3_7=0;
temp=P3;
temp=temp & 0x0f;
if (temp!=0x0f)
{
    for(i=10;i>0;i--)
    for(j=248;j>0;j--);
    temp=P3;
    temp=temp & 0x0f;

```



```

if (temp!=0x0f)
{
    temp=P3;
    temp=temp & 0x0f;
    switch(temp)
    {
        case 0x0e:
            key=0;
            break;
        case 0x0d:
            key=13;
            break;
        case 0x0b:
            key=14;
            break;
        case 0x07:
            key=15;
            break;
    }
    temp=P3;
    P1_1=~P1_1;
    if((key>=0) && (key<10))
    {
        if(keycount<6)
        {
            getps[keycount]=key;
            dispbuf[keycount+2]=19;
        }
        keycount++;
        if(keycount==6)
        {
            keycount=6;
        }
        else if(keycount>6)
        {
            keycount=6;
            keyoverflow=1;           //key overflow
        }
    }
    else if(key==12)                //delete key
    {

```

```

        if(keycount>0)
        {
            keycount--;
            getps[keycount]=0;
            dispbuf[keycount+2]=16;
        }
        else
        {
            keyoverflag=1;
        }
    }
    else if(key==15)//enter key
    {
        if(keycount!=pslen)
        {
            errorflag=1;
            rightflag=0;
            second3=0;
        }
        else
        {
            for(i=0;i<10;i++)
            {
                if(getps[i]!=ps[i])
                {
                    i=keycount;
                    errorflag=1;
                    rightflag=0;
                    second3=0;
                    goto a2;
                }
            }
            errorflag=0;
            rightflag=1;
            i=keycount;
a2:
        }
    }
    temp=temp & 0x0f;
    while(temp!=0x0f)
    {
        temp=P3;
    }

```

```

        temp=temp & 0x0f;
    }
    keyoverflag=0;
}

}

}

}

//定时器 0 中断服务子程序
void t0(void) interrupt 1 using 0
{
    /*重置定时*/
    TH0=(65536-500)/256;
    TL0=(65536-500)%6;
    flashcount++;
    if(flashcount==8)
    {
        flashcount=0;
        P0=dispcode[dispbuf[dispcount]];
        P2=dispbite[dispcount];
        dispcount++;
        if(dispcount==8)
        {
            dispcount=0;
        }
    }
    if((errorflag==1) && (rightflag==0))
    {
        bb++;
        if(bb==800)
        {
            bb=0;
            alarmflag=~alarmflag;
        }
        if(alarmflag==1)//sound alarm signal
        {
            P1_7=~P1_7;
        }
        aa++;
        if(aa==800)//light alarm signal

```

```

    {
        aa=0;
        P1_0=~P1_0;
    }
second3++;
if(second3==6400)
    {
        second3=0;
        errorflag=0;
        rightflag=0;
        alarmflag=0;
        bb=0;
        aa=0;
    }
}
else if((errorflag==0) && (rightflag==1))
    {
        P1_0=0;
        cc++;
        if(cc<1000)
            {
                okflag=1;
            }
        else if(cc<2000)
            {
                okflag=0;
            }
        else
            {
                errorflag=0;
                rightflag=0;
                P1_7=1;
                cc=0;
                oka=0;
                okb=0;
                okflag=0;
                P1_0=1;
            }
        if(okflag==1)
            {
                oka++;
            }
    }

```

```

        if(oka==2)
        {
            oka=0;
            P1_7=~P1_7;
        }
    }
    else
    {
        okb++;
        if(okb==3)
        {
            okb=0;
            P1_7=~P1_7;
        }
    }
}

if(keyoverflag==1)
{
    P1_7=~P1_7;
}
}

```

6.3 利用单片机P1 口控制直流电动机

单片机应用在工业控制系统中时，常常采用电动机来控制机械部件的平移和转动。本例将介绍如何利用单片机 P1 口来控制直流电动机。

6.3.1 实例效果说明

利用单片机 P1 口，编程输出一串脉冲，经放大后驱动小型电动机，可以改变输出脉冲的电平及持续时间，使电动机正转、反转、加速、减速、停转。

本例通过 74HC244 输入开关量数据来控制小型直流电动机的转动，实现两种转速的正转、两种转速的反转和停转。

6.3.2 74HC244 介绍

74HC244 是八输入三态缓冲器，它把 8 个输入分成 2 组，4 个一组。G=0 时，输入决定输出；G=1 时，输出为高阻态。

1. 74HC244 引脚说明

74HC244 的引脚如图 6-21 所示。

74HC244 的引脚说明如表 6-4 所示。

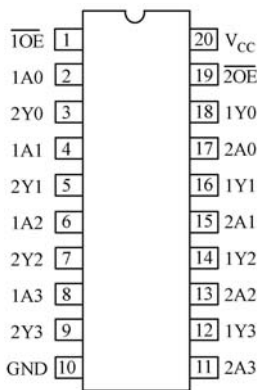


图 6-21 74HC244 引脚图

表 6-4 74HC244 引脚说明

引脚名称	说 明
V_{CC}, GND	电源, 地
$\overline{1OE}$	第一组缓冲器的使能端
1A0, 1A1, 1A2, 1A3,	第一组缓冲器的输入
1Y0, 1Y1, 1Y2, 1Y3	第一组缓冲器的输出
$\overline{2OE}$	第二组缓冲器的使能端
2A0, 2A1, 2A2, 2A3,	第二组缓冲器的输入
2Y0, 1Y1, 2Y2, 2Y3	第二组缓冲器的输出

2. 74HC244 输入输出的时序图

74HC244 输入输出的时序如图 6-22 所示, 图中显示了 74HC244 从输入到输出的传输延时以及输出的跃迁时间。

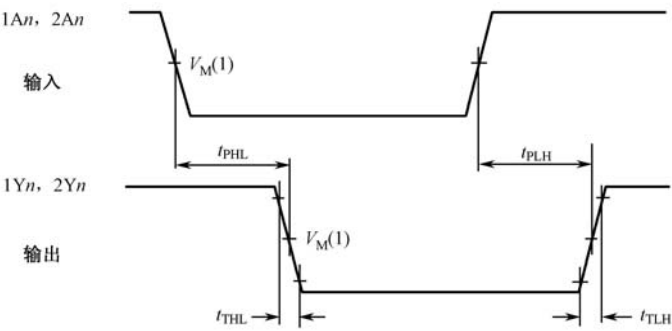


图 6-22 74HC244 输入输出的时序图

3. 74HC244 的内部框图

74HC244 的内部框图如图 6-23 所示。其中包括两组缓冲器, 第一组的使能端是引脚 1, 输入是引脚 2、4、6、8, 对应的输出是引脚 18、16、14、12; 第二组的使能端是引脚 19, 输入是引脚 11、13、15、17, 对应的输出是引脚 9、7、5、3。

各引脚的真值表如表 6-5 所示。

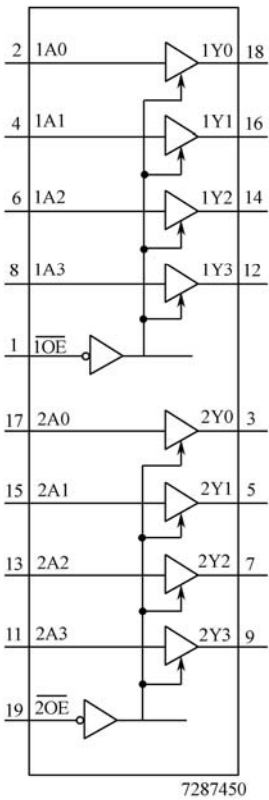


图 6-23 74HC244 的内部框图

表 6-5 74HC244 真值表

输 入		输 出
\overline{nOE}	nA_n	nY_n
L	L	L
L	H	H
H	×	Z

表中, H 代表高电平, L 代表低电平, × 代表任意, Z 代表高阻态。

6.3.3 直流电动机

直流电动机就是将直流电能转换成机械能的电动机。按励磁方式可分为自励、他励和永磁三类。

1. 直流电动机的特点及构造

直流电动机的特点如下：

- 调速性能好，可以实现均匀、平滑的无级调速，调速范围宽。即电动机在一定负载的条件下，可以根据需要人为地改变电动机的转速；
- 起动力矩大。在重负载下起动的要求均匀调节转速的机械直流电动机的优势很明显。

直流电动机分为定子与转子两部分，定子包括主磁极，机座，换向极，电刷装置等，转子包括电枢铁芯、电枢绕组、换向器、轴和风扇等。

值得注意的是，换向器是直永磁串激电动机上为了能够让电动机持续转动下去的一个部件。结构上，换向器是几个接触片围成圆型，分别连接转子上的每个触头，外边连接的两个电刷与之接触，同时只接触其中的两个。而换向极的作用是改善换向，换向极装在两主磁极之间，由铁芯和绕组构成。铁芯一般用整块钢或钢板加工而成；换向极绕组与电枢绕组串联。

2. 直流电动机的工作原理

直流电动机的工作原理如下：

- 将直流电源通过电刷接通电枢绕组，使电枢导体有电流流过；
- 电动机内部有磁场存在；
- 载流的转子（电枢）导体将受到电磁力 f 的作用（左手定则）；
- 所有导体产生的电磁力的作用于转子，使转子以 $n(r/min)$ 旋转，以便拖动机械负载。

6.3.4 硬件设计

本例的硬件原理如图 6-24 所示，其中 P1.0 口连接 74HC244 的 2A3，P1.1 口连接 74HC244 的 2A1，两个输出通过两个 74HC32 连接直流电动机电源。

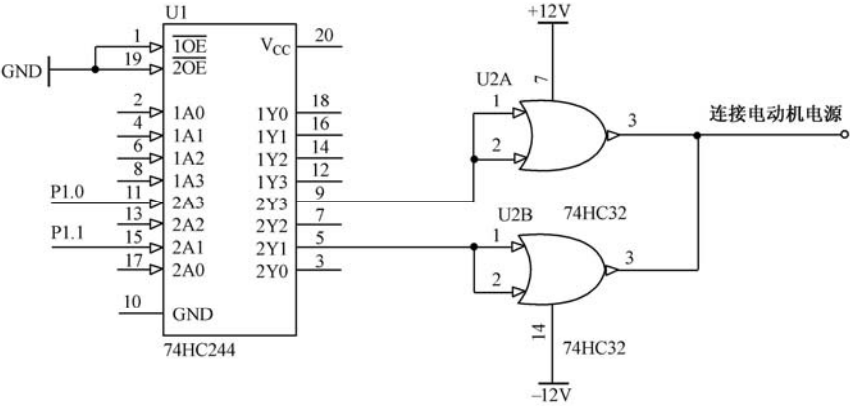


图 6-24 硬件原理图

直流电动机转动原理是：转动方向由电压控制，电压为正则正转，电压为负则反转。转速大小则是由输出脉冲的占空比来决定的，正向占空比越大则转速越快，反转时占空比越小则转速越快，如图 6-25 所示。

6.3.5 软件设计

了解了硬件的设计方案，以及电动机转向和转速的决定因素就可以设计软件部分了。

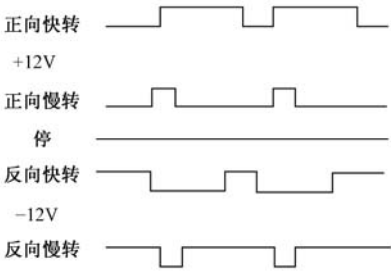


图 6-25 各状态的输出脉冲

1. 程序流程图

主程序的工作过程为：延时、正快转；延时、正慢转；延时、停；延时、负快转；延时负慢转；延时、停。全部用 for 循环来控制。其流程如图 6-26 所示。

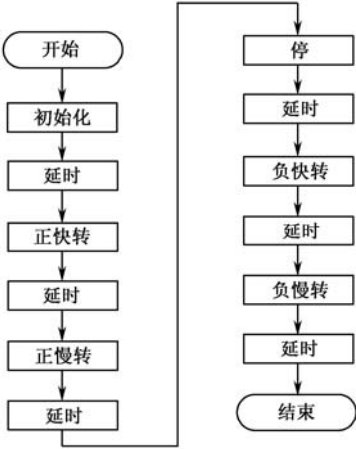


图 6-26 程序流程图

2. 程序代码

本例的程序代码说明如下：

```
#include<reg51.h>
#include<intrins.h>
#define uchar unsigned char
#define uint unsigned int

sbit P1_0=P1^0;
sbit P1_1=P1^1;
//主程序
void main ()
{
```



```

int a,m;
int b=100;
//正快转
for(a=0;a<100;a++)
{

    P1_0=1;
    for(b=0; b<50; b++)
        {
            m=0;
        }
}
P1_0=0;
for(b=0; b<10; b++)
{
    m=0;
}
//正慢转
for(a=0;a<100;a++)
{
    P1_0=1;
    for(b=0; b<10; b++)
        {
            m=0;
        }
}
P1_0=0;
for(b=0; b<50; b++)

    {
        m=0;
    }
//负快转
for(a=0;a<100;a++)
{ P1_1=1;
    for(b=0; b<50; b++)
        {
            m=0;

        }
}

```

```

    }
    P1_1=0;
    for(b=0; b<10;b++)
    {
        m=0;
    }
    //负慢转
    for(a=0;a<100;a++)
    {
        P1_1=1;
        for(b=0; b<10; b++)
        {
            m=0;
        }
    }
    P1_1=0;
    for(b=0; b<50; b++)
    {
        m=0;
    }
}

```

6.4 单片机实现智能充电器的设计

电子信息技术的快速发展使得各种各样的电子产品不断涌现，并朝着便携和小型轻量化的趋势发展，这也使得更多的电气产品采用基于电池的供电系统。事实上，几乎所有用到电池的电器设备都需要用到充电器。

目前，较多使用的电池有镍镉、镍氢、铅蓄电池和锂电池，由于它们各自的优缺点使得它们在相当长的时期内将共存发展。由于不同类型的电池的充电特性不同，通常对不同类型，甚至不同电压、容量等级的电池使用不同的充电器。

6.4.1 实例说明

随着手机向轻、薄、短、小的方向发展，对电池的需要也就相应增强，体积小而容量大的锂离子电池成为手机最小型化的必备条件。

在电池充电器领域单片机有相当广泛的应用，利用单片机的处理控制能力可以实现充电器的智能化。

本例主要介绍单片机在手机的单节锂离子电池充电器方面的智能化设计，在单片的控制下，设计具有预设、充电保护、自动断电和充电完成后报警提示功能的智能化充电器。

6.4.2 设计思路分析

要实现充电器的智能化，需要应用单片机的处理和控制的功**能**。充电的实现，是在基本的充电电压的基础上，要控制充电过程。

实现的功能模块如下：

- 实现充电器智能化控制的单片机模块；
- 采用专用的电池充电芯片实现充电过程控制模块；
- 采用电压转换芯片提供充电电压模块。

常用的充电电池有镍镉、镍氢及锂离子充电电池三种，其容量单位是 mAh。镍镉、镍氢充电电池在没有完全放电之后就进行充电，几次之后电池的容量便会减少，这种现象称为记忆效应，锂电池则没有记忆效应，所以锂电池即使在没有放电完全之后就充电，也不会影响电池的容量。

目前锂离子电池具有以下优点：

- 具有较高的能量重量比、能量体积比，输出功率大；
- 平均输出电压高（约 3.9 V），为 Ni-Cd、Ni-MH 电池的 3 倍；
- 循环性能优越，可快速充、放电，没有记忆效应；
- 充电效率高。可达 100%；
- 没有环境污染，称为绿色电池；
- 使用寿命长，可达 1 200 次左右，最长的可达 3 000 次。

手机使用的锂离子电池是使用一种精细而渗透性很强的聚乙烯薄膜隔离材料在正、负极间间隔而成的。正极包括由锂和二氧化钴组成的集电极和由铝箔膜组成的电流集电极；负极由片状碳材料组成的锂离子集电极和铜薄膜组成的电流集电极。电池内部有有机电解质溶液，还装有安全阀和 PIC 元件，保护电池在正常状态及输出短路时不受损坏。锂电池充电标准曲线如图 6-27 所示。

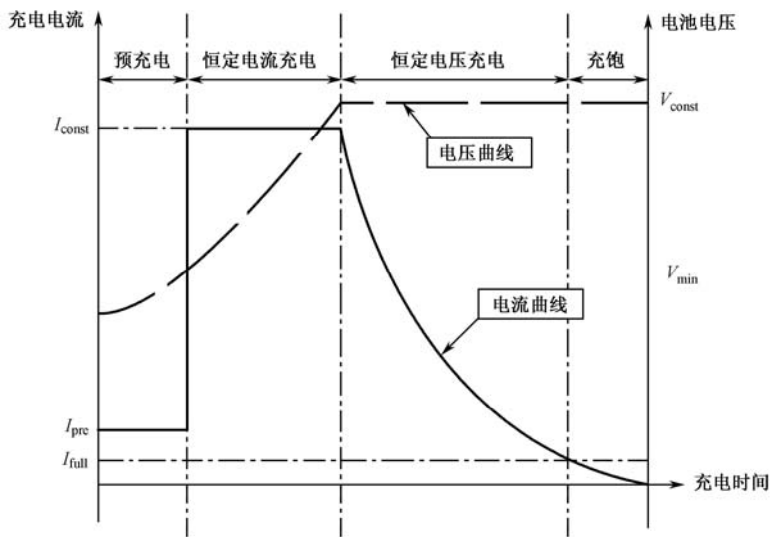


图 6-27 锂电池充电标准曲线

参数说明:

- I_{const} -恒流充电电流;
- I_{pre} -预充电电流;
- I_{full} -饱和判断电流;
- V_{const} -恒压充电电压;
- V_{min} - 预充结束电压及短路判断电压。

锂电池对充电器的要求比较高, 需要保护电路, 充电器最好带有热保护和时间保护, 为电池提供附加保护。一部好的充电器不但能在短时间内完成对电池的充电, 还可以对电池起到维护作用, 修复记忆效应。充电器的智能化可以缩短充电器的时间, 同时能够维护电池, 延长电池使用寿命。

设计一款比较科学的充电器往往要采用专用的充电控制芯片配合单片机控制的方法。专用的充电控制芯片具备较好的 $-\Delta V$ 检测, 可以检测出电池充电饱和和发出的电压变化信号, 比较精确地结束充电工作。

6.4.3 芯片介绍

为充电器选择一款合适的充电控制芯片是非常重要的, 需要依照电池类型、电池数目、电流值(电流大小决定充电时间)、充电方式(快充、慢充或可控充电)等几个参考标准。

1. MAX1898 芯片

本例选择 MAXIM 公司的 MAX1898 作为电池充电芯片, 实现手机的单节锂离子电池充电器, 充电快速并且具有较强的电池保护能力。

MAX1898 配合外部 PNP 或 PMOS 三极管可以组成完整的单节电锂离子(Li^+)电池充电器。MAX1898 提供精确的恒流/恒压充电, 电池电压调节精度为 $\pm 0.75\%$, 提高了电池性能并延长了使用寿命。

MAX1898 的主要功能特性如下。

- 使用低成本的 PNP 或 PMOS 调整元件;
- 简单、安全的线性充电方式;
- 内置检流电阻, 可编程的充电电流;
- LED 充电状态指示;
- 可编程的安全定时器;
- 可选/可调节自动重启;
- 4.5~12 V 的输入电压范围, 输入电源自动监测。

MAX1898 具有两个版本, 可对所有化学类型的 Li^+ 电池进行安全充电。电池调节电压为 4.2 V (MAX1898EUB42) 或 4.1 V (MAX1898EUB41)。两者都采用 10 引脚、超薄型 μMAX 封装。MAX1898 的引脚分布如图 6-28 所示。

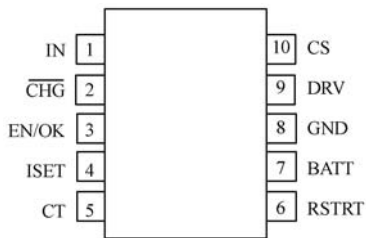


图 6-28 MAX1898 的引脚分布示意图

引脚功能说明如表 6-6 所示。

表 6-6 MAX1898 引脚功能说明

引 脚 名 称	功 能 说 明
IN	传感输入，检测输入的电压或电流
$\overline{\text{CHG}}$	充电状态指示脚，同时驱动 LED
EN/OK	使能输入脚/输入电源“好”输出指示脚。EN 为输入脚，可以通过输入禁止芯片工作；OK 为输出脚，用于指示输入电源是否与充电器连接
ISET	充电电流调节引脚。通过在地之间串联一个电阻来设置最大充电电流
CT	安全充电时间设置引脚，通过接一个时间电容来设置最大充电电流
RSTRT	自动重新启动控制引脚，当此引脚直接接地时，如果电池电压掉至基准电压阈值以下 200 mV，将重新开始一个充电周期。
BATT	电池传感输入引脚，接单个 Li ⁺ 电池的正极，此引脚与地之间需要接一个大电解电容
GND	接地端
DRV	外部三极管驱动器，接三极管的基极
CS	电流传感输入，接三极管的发射极

MAX1898 芯片内部电路包括输入电流调节器、电压检测器、充电电流检测器、定时器、温度检测器和主控器。MAX1898 的内部结构如图 6-29 所示。

使用 MAX1898 时，充电电流由用户设定，采用内部检流，无需外部检流电阻。MAX1898 提供用于监视充电状态的输出、输入电源是否与充电器连接的输出指示和充电电流指示。其他功能包括关断控制、可选的充电周期重启、无需重新上电、可选的充电终止安全定时器和过放电池的 low 电流预充。

MAX1898 具有更高的集成度，在更小的尺寸内集成了更多的功能，尽可能多地覆盖了基本应用电路，只需少数外部元件。MAX1898 的典型充电电路如图 6-30 所示。

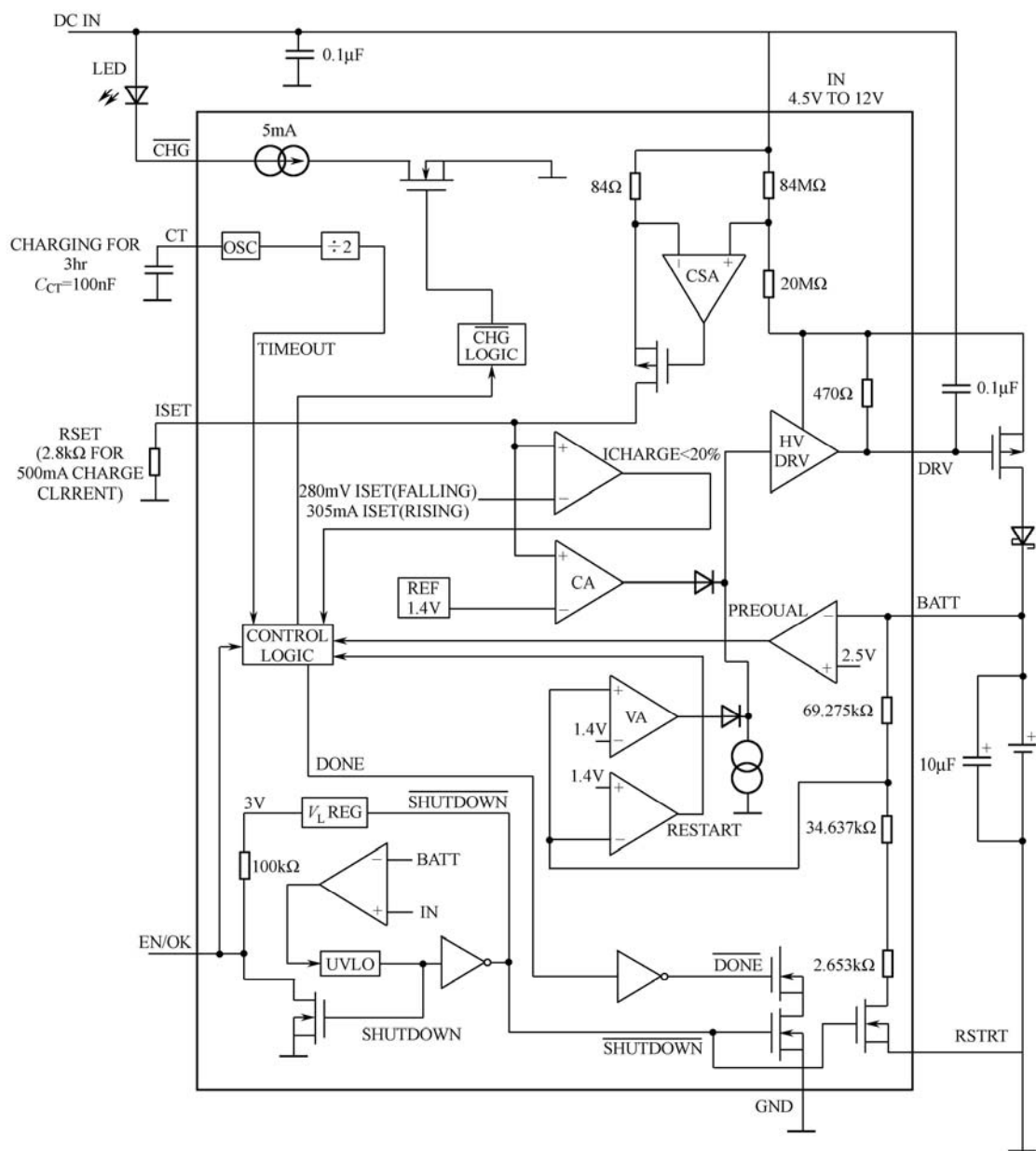


图 6-29 MAX1898 的内部结构示意图

电路及应用的具体说明如下。

① 输入电压的范围为 4.5~12 V。锂电池要求的充电方式是恒流恒压方式，一般采用滞留电源外加变压器作为恒流恒压源。当充电电源和电池在正常的工作范围内时插入电池将启动一次充电过程。MAX1898 能够自动监测充电电源，没有电源时自动关掉以减少电池的漏电。

② 通过外接的场效应管提供锂电池的充电接口。启动快充后，打开外接的 P 型场效应管，电池的电压达到设定的门限时进入脉冲充电方式，P 型场效应管打开的时间会越来越短。

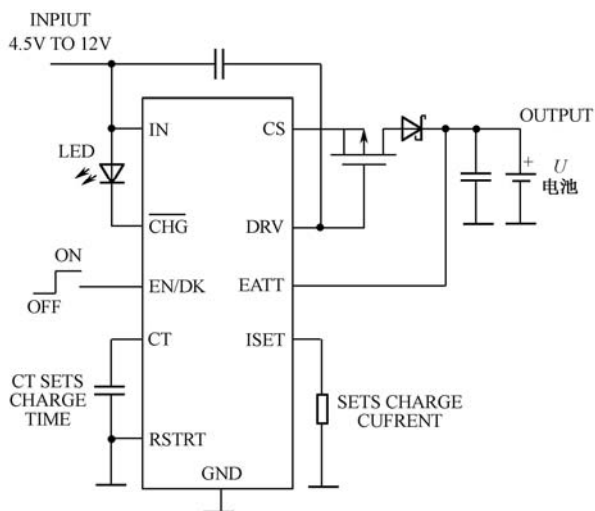


图 6-30 MAX1898 的典型充电电路

- ③ 通过外接的电容器来设置充电时间，充电时间指的是快充时的最大充电时间。一般情况下常取外接电容值为 $100\ \mu\text{F}$ ，即快充最大充电时间为 3 小时左右。
- ④ 在限制电流的模式下，通过外接的电阻来设置最大充电电流。
- ⑤ 充电结束时，LED 灯会周期性地闪烁，具体的闪烁含义如表 6-7 所示。

表 6-7 MAX1898 充电电路 LED 闪烁含义

充 电 状 态	LED 指示灯
电池或充电器没有安装	灭
预充或快充	亮
充电结束	灭
充电出错	以 1.5Hz 频率闪烁

2. LM7805 芯片

由于充电器外部为 +12 V 电压供电，需要通过电压转换芯片将 +12 V 电压转换为 +5 V 电压，这里选用国家半导体公司生产的三端固定稳压转换芯片 LM7805 来完成电压转换。

LM7805 用于将输入的电压稳压为 +5 V 后提供给有关电路，其引脚分布如图 6-31 所示。

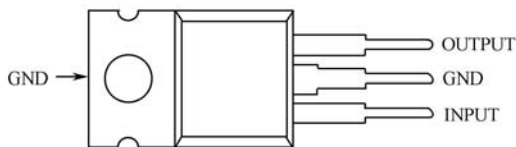


图 6-31 LM7805 的引脚分布

LM7805 的引脚的功能说明如表 6-8 所示。

表 6-8 LM7805 的引脚的功能说明

引 脚 名 称	功 能 说 明
INPUT	需要转换的+12 V 电压输入端
GND	接地端
OUTPUT	+5 V 电压输出端

在使用 LM7805 之前，需要了解它的基本工作参数以及各参数的意义。

- 输出电压：输出电压稳定在 +5 V；
- 线性调整率：在常温时输出 500 mA 电流的情况下，输入电压在 7~25 V 之间变化的时候，输出电压的变化典型值为 3 mV，最大值为 50 mV；
- 负载调整率：在负载变化在 5~1.5 A 时，输出电压的变化范围在 10~50 mV；
- 静态电流：为驱动大功率调整管所必须的，它并不流向负载，而是直接流向地。在负载小于 1A 的情况下，静态电流为 8 mA；
- 静态电流变化量：在负载变化范围在 5 mA~1 A 时，静态电流的变化为 0.5 mA（静态电流增大）；
- 输出噪声电压：在 10 Hz~100 kHz 频率范围内，输出噪声电压为 40 μ V；
- 纹波抑制比：频率越高，纹波抑制比一般会越小。在负载电流小于 1 A 的情况下，120 kHz 点的纹波抑制比最小为 62 dB，典型值为 80 dB；
- 电压降下量：负载为 1 A 的时候，电压降下量为 2 V，也就是说要维持 +5 V 的输出，输入电压必须在 7 V 以上；
- 输出抵抗：同频率有关，频率越高，输出电抗会有所增大；在 1 kHz 的时候，输出抵抗为 8 m Ω ；
- 短路保护电流：电流为 2.1 A；
- 尖峰输出保护电流：该电流为 2.4 A；
- 维持输出所需要的最小输入电压：为了保证输出性能，输入电压必须大于该值，LM7805 中该电压为 7.5 V。

3. 6N137 芯片

为了降低电源的干扰，保持电路的稳定，完成电压转换之后，需要经过一次光耦模块的处理，通过单片机对光耦模块的控制，可以及时关断充电电源。这里采用的光耦模块为 6N137 光耦合器。

6N137 光耦合器是一款用于单通道的高速光耦合器，主要功能特性如下。

- 高达 10 Mb/s 的转换速率；
- 高达 10 kV/ μ s 的摆率；
- 扇出系数为 8；
- 逻辑电平输出；
- 集电极开路输出。

6N137 光耦合器的内部结构以及引脚分布如图 6-32 所示，6N137 光耦合器的引脚功能

如表 6-9 所示。

表 6-9 6N137 光耦合器引脚功能说明

引 脚 名 称	功 能 说 明
NC	悬空
+、-	发光二极管的正、负极
GND	接地端
V _O	输出端
EN	使能端，高电平有效。EN=0 时，无论有无输入，输出都为高；不使用时，悬空即可
V _{CC}	电源输入端

6N137 光耦合器内部有一个 850 nm 波长 AlGaAs LED 和一个集成检测器组成，其检测器由一个光敏二极管、高增益线性运放及一个集电极开路的三极管组成。具有温度、电流和电压补偿功能。

6N137 光耦合器工作时，信号从引脚 2 和引脚 3 输入，发光二极管发光，经片内光通道传到光敏二极管，反向偏置的光敏管光照后导通，经电流-电压转换后送到与门的一个输入端，与门的另一个输入为使能端，当使能端为高电平时与门输出高电平，经输出三极管反向后光电隔离器输出低电平。当输入信号电流小于触发阈值或使能端为低时，输出高电平，但这个逻辑高电平是集电极开路的，可针对接收电路加上拉电阻或电压调整电路。

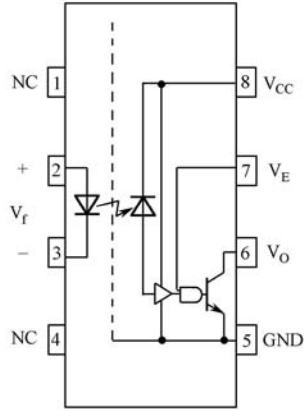


图 6-32 6N137 光耦合器内部结构及引脚分布

6.4.4 硬件电路设计

硬件电路由单片机电路、电压转换及光耦合隔离电路、充电控制电路三部分组成。通过单片机的控制实现预充、快充、满充、断电、报警充电过程。

单片机部分的电路原理如图 6-33 所示。

图 6-33 中单片机芯片为 Atmel 公司的 AT89C52（U1），B1 为蜂鸣器，单片机的 P0.0 口输出控制光耦器件，可以在需要时及时关掉充电电源。

基于 MAX1898 的智能充电器电路电压转换、光耦合隔离电路部分的原理如图 6-34 所示。

图 6-34 中，电压转换芯片 LM7805 将 +12 V 输入电压转化为固定的 +5 V 输出，光耦隔离芯片 6N137 将 +5 V 电压隔离后输出。为了应用单片机适时的控制充电电源的关闭，将光耦隔离芯片 6N137 的第 2 引脚与单片机的 P0.0 口相连。

基于 MAX1898 的智能充电器电路充电控制部分电路如图 6-35 所示。

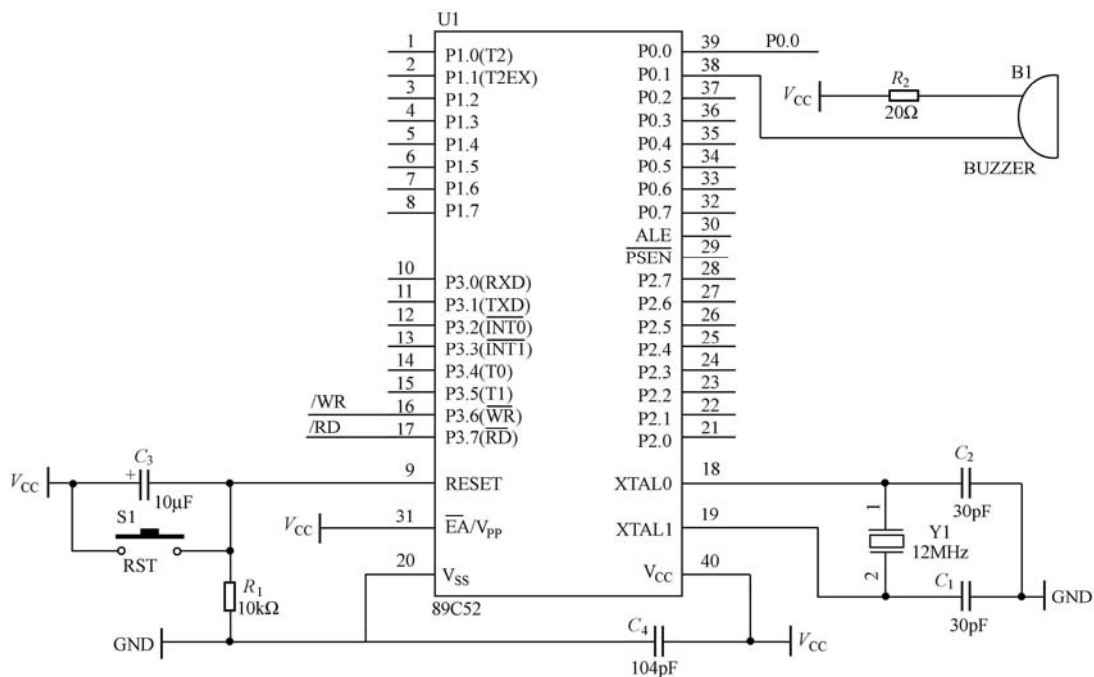


图 6-33 单片机部分的电路原理图

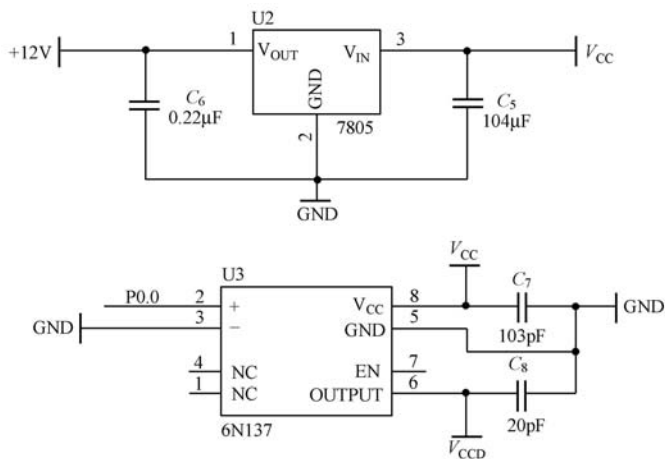


图 6-34 电压转换及光耦隔离电路原理图

图 6-35 中，充电状态输出引脚 $\overline{\text{CHG}}$ 经过反相器 74LS04 后与单片机的 P3.2 口相连，触发外部中断。D2 为红色发光二极管，电源接通时亮；D1 为绿色发光二极管，处于充电状态时亮。 R_4 为设置充电电流的电阻，阻值为 $2.8\text{ k}\Omega$ ，设置最大充电电流为 500 mA ； C_{11} 为设置充电时间的电容，容值为 $100\text{ }\mu\text{F}$ ，设置最大充电时间为 3 小时。

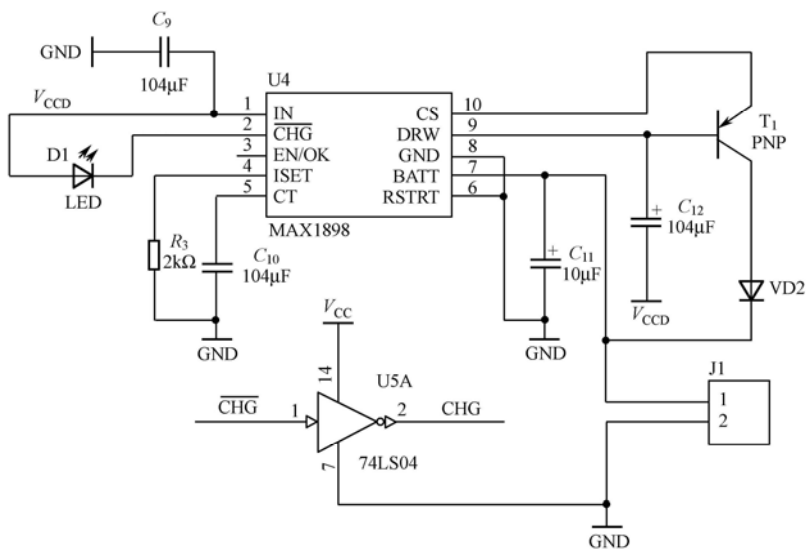


图 6-35 充电控制部分电路图

6.4.5 软件设计

当 MAX1898 完成充电时，引脚 $\overline{\text{CHG}}$ 会产生由低到高的跳变，该跳变引起单片机的 INT0 中断。 $\overline{\text{CHG}}$ 输出为高电平时有三种情况，分别为无电池或无充电输入、充电完毕、充电出错（ $\overline{\text{CHG}}$ 以 1.5 Hz 频率反复跳变）。前两种情况下，单片机可以直接控制光耦模块切断充电电源，程序中需要区别对待的是充电出错时的情况，如果判断出不是充电出错，单片机控制 P2.0 口切断电源，控制 P2.1 口启动蜂鸣器报警。

1. 设计流程

单片机控制 MAX1898 智能充电器工作的程序流程如图 6-36 所示。

2. 程序说明

单片机控制 MAX1898 智能充电器工作的程序代码以及说明如下：

```
#include <reg52.h>
#define uchar unsigned char
#define uint unsigned int

uint t_c;
uint int0_c;

void main()
{
    init();
    while(1);
}
```

```
}
```

```
//初始化
```

```
void init()
```

```
{
```

```
    EA = 1;                //开 CPU 中断
    PT0 = 1;               // T0 中断优先级设置
    TMOD = 0x01;           //模式 1，T0 为 16 位计数器
    ET0 = 1;               //开 T0 中断
```

```
    IT0 = 1;               // 外部中断，边沿触发
    EX0 = 1;               //开外部中断
```

```
    GATE = 1;              //光耦合器正常输出电压
    BP = 1;                //关蜂鸣器
```

```
    int0_c = 0;            //计数器清零
```

```
}
```

```
//定时器 0 中断服务子程序
```

```
void timer0() interrupt 1 using 1
```

```
{
```

```
    TR0 = 0;               //停止计数
    TH0 = -5000/256;        //设置计数初值
    TL0 = -5000%256;
    t_c++;
```

```
    if (t_c>600)            //第一次外部中断产生后 3 s
```

```
    {
```

```
        if (int0_c==1)      //没有出现第二次外部中断，说明充电完毕
```

```
        {
```

```
            GATE = 0;        //关闭电源
            BP = 0;           //打蜂鸣器
```

```
        }
```

```
    else                    //充电出错
```

```
    {
```

```
        GATE = 1;
        BP = 1;
```

```
    }
```

```
    ET0 = 0;               //关 T0
    EX0 = 0;               //关外部中断 0
```

```

        int0_c = 0;
        t_c = 0;
    }
    else
        TR0 = 1;          //启动 T0 计数
}

//外部中断 0 服务子程序
void int0() interrupt 0 using 1
{
    if (int0_c==0)
    {
        TH0 = -5000/256;    //定时 5 ms
        TL0 = -5000%256;
        TR0 = 1;           //启动计数器计数
        t_c = 0;           //计数器清零
    }
    int0_c++;
}

```

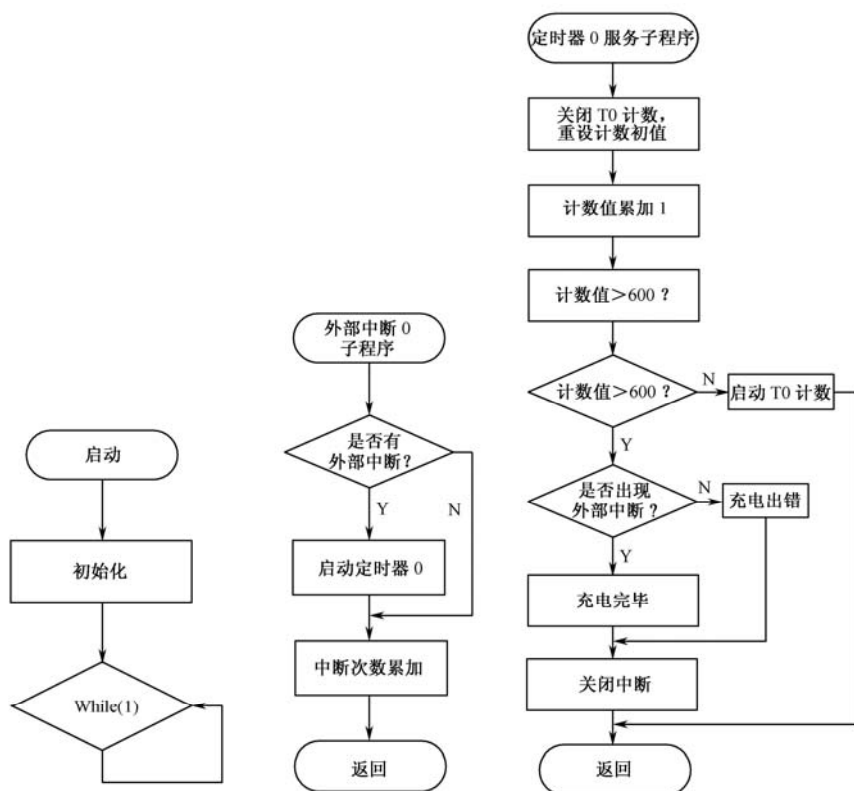


图 6-36 单片机控制智能充电器工作的程序流程

6.5 基于DS12C887 芯片的实时时钟日历显示

实时时钟（RTC）作为系统同步或时间标志已被广泛应用于各种电子产品，利用 Dallas Semiconductor 公司提供的多种类型的 RTC 芯片，用户在设计中可方便地针对具体应用来选择相应的芯片。

6.5.1 实例说明

本例通过 DS12C887 实时时钟日历芯片在 C51 单片机中设置、获取并显示实时时钟和日历，并可以长期保存时间信息。

6.5.2 DS12C887 芯片说明

DS12C887 实时时钟芯片的功能非常丰富，可以直接代替 IBM PC 上的时钟日历芯片 DS12887。DS12C887 的引脚和 MC146818B、DS12887 相兼容。

DS12C887 能够自动产生世纪、年、月、日、时、分、秒等时间信息，而且其内部又增加了世纪寄存器，从而利用硬件电路解决了“千年”问题。DS12C887 自带锂电池，即使外部掉电，其内部的时间信息也可以保持 10 年之久。

一天之内的时间记录，分为 12 小时制和 24 小时制两种模式。在 12 小时制模式中，利用 AM 和 PM 区分上午和下午。时间的表示方法包括使用二进制数表示和使用 BCD 码表示两种。DS12C887 中带有 128 B RAM，其中有 11 B RAM 用来存储时间信息，4 B RAM 用来存储 DS12C887 的控制信息，称为控制寄存器，113 B 通用 RAM 用户使用；此外用户还可对 DS12C887 进行编程以实现多种方波输出，并可对其内部的三路中断通过软件进行屏蔽。

DS12C887 的引脚如图 6-37 所示。

1. 结构框图和引脚功能

DS12C887 芯片的内部结构如图 6-38 所示，通过该图可以对该芯片的工作原理有一个更深入的了解。

由图 6-38 可知，电源、时钟/日历信息、寄存器以及总线接口等部分共同实现了 DS12C887 的实时时钟日历功能。

DS12C887 引脚功能如表 6-10 所示。

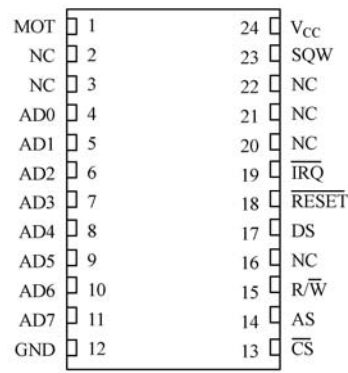


图 6-37 DS12C887 引脚图

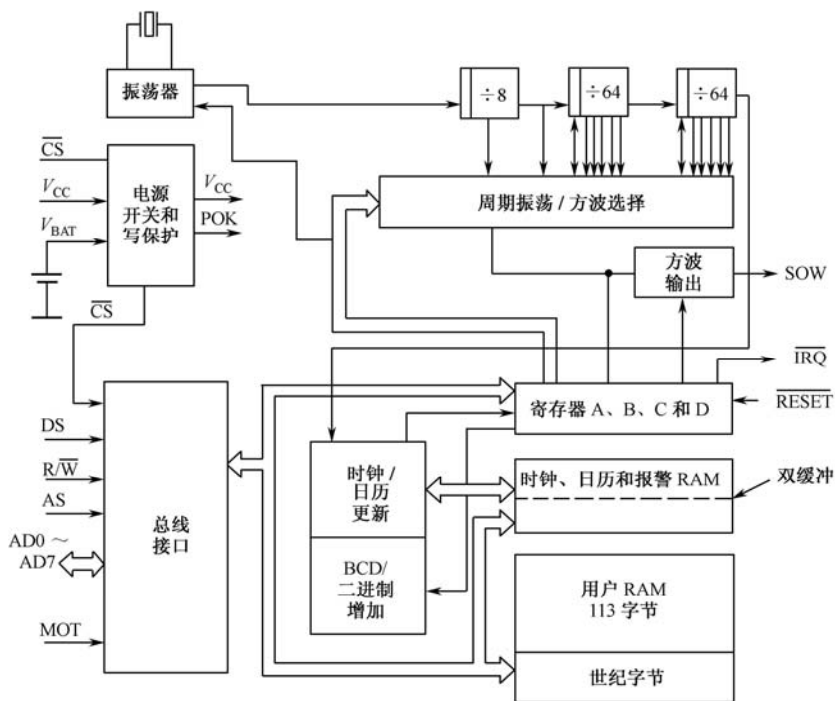


图 6-38 内部结构框图

表 6-10 DS12C887 引脚功能

引脚名称	说明
AD0~AD7	复用地址数据总线，该总线采用时分复用技术，在总线周期的前半部分，出现在 AD0~AD7 上的是地址信息，可用以选通 DS12C887 内的 RAM，总线周期的后半部分出现在 AD0~AD7 上的数据信息
NC	空引脚，无连接
MOT	模式选择引脚，DA12C887 有两种工作模式，即 Motorola 模式和 Intel 模式，当 MOT 接 V _{CC} 时，选用的工作模式是 Motorola 模式，当 MOT 接 GND 时，选用的是 Intel 模式
\overline{CS}	片选输入，低电平有效
AS	地址选通输入引脚，在进行读/写操作时，AS 的上升沿将 AD0~AD7 上出现的地址信息锁存到 DS12C887 上，而下一个下降沿清除 AD0~AD7 上的地址信息，不论是否有效，DS12C887 都将执行该操作
R/ \overline{W}	读/写输入端，该引脚有 2 种工作模式，当 MOT 接 V _{CC} 时，R/ \overline{W} 工作在 Motorola 模式。此时，该引脚的作用是区分进行的是读操作还是写操作，当 R/ \overline{W} 为高电平时为读操作，R/ \overline{W} 为低电平时为写操作；当 MOT 接 GND 时，该脚工作在 Intel 模式，此时该作为写允许输入，即写使能
DS	数据选通
\overline{RESET}	复位引脚，不影响时钟、日历或 RAM。上电时，复位引脚可以保持低电平一段时间，以使供电应趋于稳定
\overline{IRQ}	中断请求输入，低电平有效，该引脚有效时对 DS12C887 内的时钟、日历和 RAM 中的内容没有任何影响，仅对内部的控制寄存器有影响，在典型的应用中， \overline{RESET} 可以直接接 V _{CC} ，以保证 DS12C887 在掉电时，其内部控制寄存器不受影响

引 脚 名 称	说 明
SQW	方波输出引脚，当供电电压 $V_{CC}>4.25\text{ V}$ 时，SQW 引脚可进行方波输出，此时用户可以通过对控制寄存器编程来得到 13 种方波信号的输出
V_{CC}	直流电源，其中 V_{CC} 接+5 V 输入。当 V_{CC} 输入为+5 V 时，用户可以访问 DS12C887 内 RAM 中的数据，并可对其进行读、写操作；当 V_{CC} 的输入 \leq +4.25 V 时，禁止用户对内部 RAM 进行读、写操作，此时用户不能正确获取芯片内的时间信息；当 $V_{CC} \leq 3\text{ V}$ 时，DS12C887 会自动将电源转换到内部自带的锂电池上，以确保内部的电路能够正常工作
GND	电源地

2. 存储功能

DS12C887 内有 11 B 的 RAM 用来存储时间信息，其中 4 B 用来存储控制信息，如表 6-11 所示。而且，在表 6-10 中可以看出 DS12C887 内部有 4 个控制寄存器，用户都可以在任何时候对其进行访问来对 DS12C887 进行控制操作。

表 6-11 地址及取值范围

地 址	功 能	取值范围（十进制）	取 值 范 围	
			二进制	BCD 码
0	秒	0~59	00~3B	0~59
1	秒闹铃	0~59	00~3B	0~59
2	分	0~59	00~3B	0~59
3	分闹铃	0~59	00~3B	0~59
4	12 小时模式	0~12	01~0C AM, 81~8C PM	01~12 AM, 81~92 PM
	24 小时模式	0~23	00~17	00~23
5	时闹铃，12 小时制	0~12	01~0C AM, 81~8C PM	01~12 AM, 81~92 PM
	时闹铃，24 小时制	0~23	00~17	00~23
6	星期几（星期天=1）	0~7	01~07	01~07
7	日	0~31	01~1F	01~31
8	月	0~12	01~0C	01~12
9	年	0~99	0~63	0~99
10	控制寄存器 A			
11	控制寄存器 B			
12	控制寄存器 C			
13	控制寄存器 D			
50	世纪	0~99	NA	19, 20

(1) 控制寄存器 A

控制寄存器 A 不受复位的影响，UIP 位为只读位，其他各位均可读/写，寄存器 A 的控制字的格式如表 6-12 所示。

UIP 位：更新周期标志位，为 1 时，表示芯片正处于或将开始更新周期，此时程序不允

许读/写时标寄存器；为 0 时，表示至少在 244 μ s 后才开始更新周期，此时程序可读芯片内时标寄存器。

表 6-12 控制寄存器 A 控制字的格式

BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
UIP	DV2	DV1	DV0	RS3	RS2	RS1	RS0

DV0~DV2：芯片内部振荡器 RTC 控制位。当芯片解除复位状态，并将 010 写入 DV0~DV2 后，另一个更新周期将在 500 ms 后开始。因此，在程序初始化时可利用这三位使芯片在设定的时间里开始工作。

RS0~RS3：周期中断可编程方波输出速率选择位。不同的组合可以产生不同的输出，程序可以通过设置寄存器 B 的 SQWF 和 PIE 位控制是否允许周期中断方波输出。控制寄存器 A 输出速率选择位如表 6-13 所示。

表 6-13 控制寄存器 A 输出速率选择位

寄存器 A 输出速率选择位				32 768 Hz 时基	
RS3	RS2	RS1	RS0	中断周期	SQWF 输出频率
0	0	0	0	无	无
0	0	0	1	3.96625 ms	256 Hz
0	0	1	0	7.8125 ms	128 Hz
0	0	1	1	122.07 μ s	8.192 kHz
0	1	0	0	244.141 μ s	4.096 kHz
0	1	0	1	488.281 μ s	2.048 kHz
0	1	1	0	976.562 μ s	1.024 kHz
0	1	1	1	1.953125 ms	512 Hz
1	0	0	0	3.90625 ms	256 Hz
1	0	0	1	7.812 ms	128 Hz
1	0	1	0	15.625 ms	64 Hz
1	0	1	1	31.25 ms	32 Hz
1	1	0	0	62.5 ms	16 Hz
1	1	0	1	125 ms	8 Hz
1	1	1	0	250 ms	4 Hz
1	1	1	1	500 ms	2 Hz

(2) 控制寄存器 B

控制寄存器 B 用于控制芯片的工作状态，允许读/写。控制寄存器 B 控制字的格式如表 6-14 所示。

表 6-14 控制寄存器 B 控制字的格式

BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
SET	PIE	AIE	UIE	SQWE	DM	24/12	DSF

SET 位：为 0 时，芯片处于正常工作状态，每秒产生一个更新周期来更新时标寄存器。为 1 时，芯片停止工作，程序在此期间可初始化芯片的各个时标寄存器。

PIE、AIE、UIE 位：分别为周期中断、报警中断、更新周期结束中断允许位。为“1”时，允许芯片发相应的中断。

SQWE 位：方波输出允许位。SQWE=1 时，按寄存器 A 输出速率选择位所确定的频率输出方波；SQWE=0 时，SQW 引脚保持低电平。

DM 位：时标寄存器用于十进制 BCD 码表示或用二进制表示格式选择位。DM=0 时，为十进制 BCD 码；DM=1 时，为二进制码。

24/12 位：24/12 小时模式设置位。24/12 位=1 时，为 24 小时工作模式；24/12=0 时，为 12 小时工作模式。

DSE 位：夏令时服务位。DSE=1 时，夏时制设置有效，夏时制结束可自动刷新恢复时间；DSE=0，无效。

(3) 控制寄存器 C

控制寄存器 C 的控制字的格式如表 6-15 所示。该寄存器的特点是程序访问该寄存器后，寄存器的内容将自动清零，从而使 IRQF 标志位变为高电平，否则，芯片将无法向 CPU 申请下一次中断。

表 6-15 控制寄存器 C 控制字的格式

BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
IRQF	PF	AF	UF	0	0	0	0

IRQF 位：中断申请标志位。逻辑表达式为： $IRQF=PF \cdot PIE+AF \cdot AIE+UF \cdot UIE$ 。当 IRQF 位为 1 时，引脚将变低电平引发中断申请。

PF、AF、UF 位：分别为周期中断、报警中断、更新周期结束中断标志位。只要满足各中断的条件，相应的中断标志位将置 1。

BIT3~BIT0：未定义的保留位，读出值始终为 0。

(4) 控制寄存器 D

控制寄存器 D 为只读寄存器，寄存器 D 的控制字的格式如表 6-16 所示。

表 6-16 控制寄存器 D 控制字的格式

BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
VRT	0	0	0	0	0	0	0

VRT 位：芯片内部 RAM 与寄存器内容有效标志位。为 1 时，指芯片内部 RAM 和寄存器内容有效。读该寄存器后，该位将自动置 1。

BIT6~BIT0 位：保留位，读出的数值始终为 0。

6.5.3 硬件电路图设计

基于 DS12C887 芯片的实时时钟日历显示的单片机控制部分原理如图 6-39 所示，其

中，P1 口用做数码管的段码接口，P2.1~P2.4 口用做数码管的位码接口。

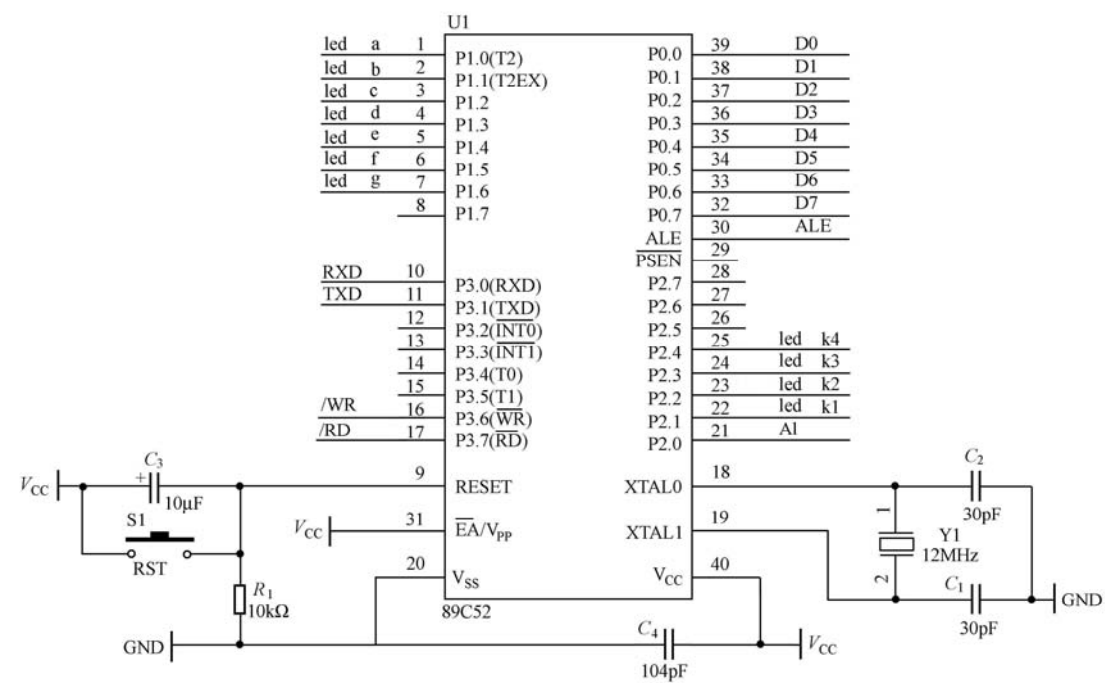


图 6-39 实时时钟日历显示的单片机控制部分原理图

图 6-40 是基于 DS12C887 芯片的实时时钟日历显示的片选及时钟芯片原理图，其中 U3 为 6 反相器 74LS04，U2 为 DS12C887 日历时钟芯片。

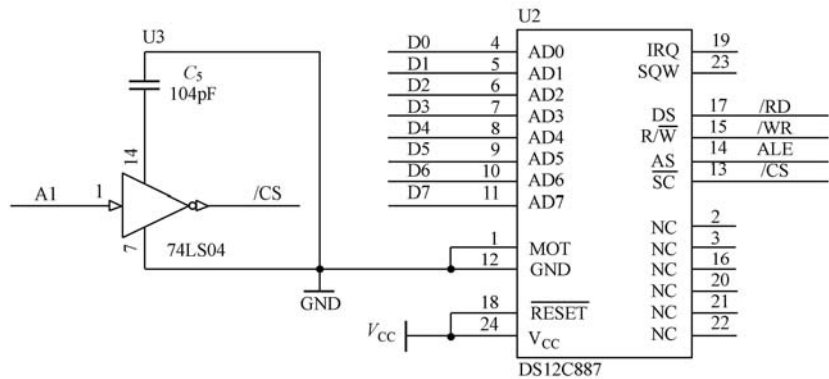


图 6-40 实时时钟日历显示的片选及时钟芯片原理图

6.5.4 软件设计

单片机控制时钟日历芯片 DS12C887，然后通过寄存器设置和读取信息。

1. 程序流程

程序流程如图 6-41 所示。

2. 程序说明

本例通过在主程序中设置出写入时间，然后显示出来。
具体程序说明如下：

```
#pragma small          //预处理，根据优先级大小先后处理
#include<reg52.h>
#include<math.h>
#include<absacc.h>
#include<stdio.h>
#define uchar unsigned char
#define uint   unsigned int

//DS12C887 寄存器地址定义
#define DS12C887_S   XBYTE [0x3f00]//秒
#define DS12C887_SA XBYTE [0x3f01]//秒报警
#define DS12C887_M   XBYTE [0x3f02]//分
#define DS12C887_MA XBYTE [0x3f03]//分报警
#define DS12C887_H   XBYTE [0x3f04]//时
#define DS12C887_HA XBYTE [0x3f05]//时报警
#define DS12C887_W   XBYTE [0x3f06]//星期
#define DS12C887_DAY XBYTE [0x3f07]//日
#define DS12C887_MT  XBYTE [0x3f08]//月
#define DS12C887_Y   XBYTE [0x3f09]//年
#define DS12C887_A   XBYTE [0x3f0a]//寄存器 A，A.7—只读判断忙闲
#define DS12C887_B   XBYTE [0x3f0b]//寄存器 B，B.7—控制工作状态：0 正常、1 停止
//B.2—BCD 码或二进制选择：0 表示二进制
//B.1—24/12 小时模式选择：1 表示 24 小时进制
#define DS12C887_C   XBYTE [0x3f0c]//寄存器 C
#define DS12C887_D   XBYTE [0x3f0d]
//寄存器 D 只读，读取该寄存器可使设置的时间和模式生效

//DS12C887 相关变量定义
uchar year;
uchar month;
uchar day;
uchar hour;
uchar minute;
uchar second;

//时钟判断是否空闲
void clock_judge_busy(void)
```

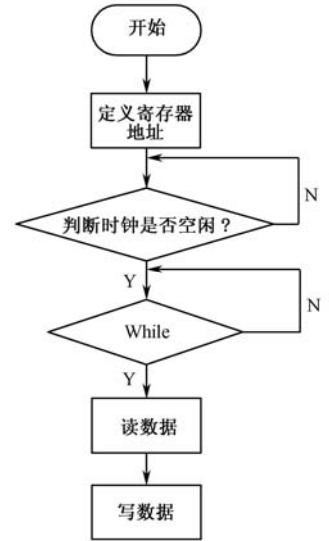


图 6-41 程序流程图

```

{ while( DS12C887_A & 0x80 );    //忙则循环等待
}

//时钟读数据
void clock_read_data(void)
{
    clock_judge_busy();
    year = DS12C887_Y;
    clock_judge_busy();
    month = DS12C887_MT;
    clock_judge_busy();
    day = DS12C887_DAY;
    clock_judge_busy();
    hour = DS12C887_H;
    clock_judge_busy();
    minute = DS12C887_M;
    clock_judge_busy();
    second = DS12C887_S;
}

//时钟写数据
void clock_write_data(void)
{ uchar i;
    DS12C887_B = 0x82;
    DS12C887_Y = year;
    DS12C887_MT = month;
    DS12C887_DAY = day;
    DS12C887_H = hour;
    DS12C887_M = minute;
    DS12C887_S = second;
    DS12C887_A = 0x20;
    i= DS12C887_C;
    i= DS12C887_D;
    DS12C887_B = 0x02;
}

//主程序
void main()
{
    year    = 8; //写入时间： 2008 年 4 月 1 日 12 时 34 分 38 秒
    month   = 4;
    day     = 1;

```

```
hour    = 12;
minute = 34;
second = 38;
clock_write_data();
```

```
clock_read_data();//读取的时间数据在变量 year/month/day/hour/minute/second 中
```

```
}
```

6.6 单片机实现步进式PWM信号输出

脉宽调制（PWM）技术最初是在无线电技术中用于信号的调制，后来在电动机调速中得到了很好的应用，形成了独特的 PWM 控制技术。

6.6.1 实例说明

本例介绍一种新型 PWM 输出的方式，它用单片机 89C52 作为主控部分，用 8254 可编程定时/计数器来实现步进式 PWM 的输出，具有分辨率高、反应速度快及占用 CPU 时间少的优点。

本例的功能模块可以分为以下几个部分：

- 单片机 89C52：主控部分，控制外部计数芯片实现脉冲计数；
- 外部计数器电路：计数芯片与单片机的接口电路，以及外围电路；
- 单片机程序：控制技术芯片正常工作。

6.6.2 设计思路分析

在进行单片机实现步进式 PWM 信号输出设计之前，了解 PWM 控制技术的知识是至关重要的。

1. PWM基本原理及其实现方法

PWM 是通过控制固定电压的直流电源开关频率，从而改变负载两端的电压，进而达到控制要求的一种电压调整方法。PWM 可以应用在许多方面，如电动机调速、温度控制、压力控制等。

（1）PWM 基本原理

脉冲宽度调制波通常由一系列占空比不同的矩形脉冲构成，其占空比与信号的瞬时采样值成比例。图 6-42（a）和（b）分别为脉冲宽度调制系统的原理框图和波形图。

该系统有一个比较器和一个周期为 T_s 的锯齿波发生器组成。语音信号如果大于锯齿波信号，比较器输出正常数 A ，否则输出 0。因此，从图 6-42（a）中可以看出，比较器输出一系列下降沿调制的脉冲宽度调制波。

通过对图 6-42（b）的分析可以看出，生成的矩形脉冲的宽度取决于脉冲下降沿时刻 t_k 时的语音信号幅度值。因而，采样值之间的时间间隔是非均匀的。在系统的输入端插入一个

采样保持电路可以得到均匀的采样信号，但是对于实际中 $t_k - kT_s \ll T_s$ ，均匀采样和非均匀采样差异非常小。脉冲宽度调制波可以直接通过低通滤波器进行解调。

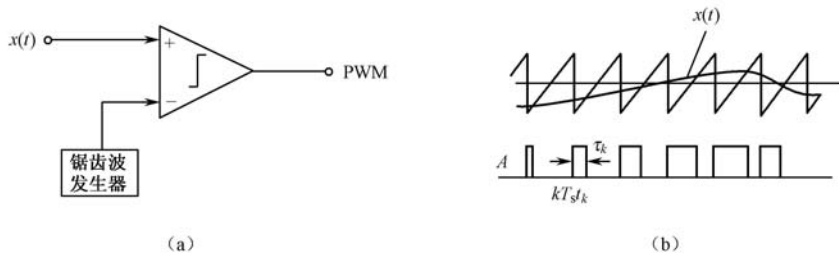


图 6-42 脉冲宽度调制系统的原理框图和波形图

(2) 实现方法

实现数字脉冲宽度调制器的基本思想如图 6-43 所示。

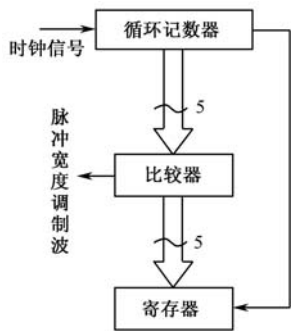


图 6-43 数字脉冲宽度调制器的构成

由图 6-43 可知，在时钟脉冲的作用下，循环计数器的 5 位输出逐次增大。5 位数字调制信号用一个寄存器来控制，不断与循环计数器的输出进行比较，当调制信号大于循环计数器的输出时，比较器输出高电平，否则输出低电平。循环计数器循环一个周期后，向寄存器发出一个使能信号 EN，寄存器送入下一组数据。在每一个计数器计数周期，由于输入的调制信号的大小不同，比较器输出端输出的高电平个数不一样，因而产生出占空比不同的脉冲宽度调制波。

奇偶序列的产生方法是将计数器的最后一位作为比较数据的最低位，在一个计数周期内，前半周期计数器输出最低位为 0，其他高位逐次增大，则产生的数据即为偶数序列；后半周期输出最低位为 1，其余高位依次减小，产生的数据为依次减小的偶序列。

具体电路如图 6-44 所示。

一般情况下，调节脉冲宽度信号的脉宽有两种方法，一种方法是采用模拟电路中的调制方法，另外一种方法是脉冲计数法。对于一般电动机控制，由于滤波频率较低、滤波精度要求高和滤波电路的参数不易调整等原因，采用第一种方法在控制电压变化时滤波的实现存在较大的困难。因此，本例主要介绍单片机控制实现的脉冲计数法。

2. 芯片性能介绍

一般的 C51 单片机的内部定时器时钟最大可以取 24 MHz，但指令周期为 0.5 μ s，计数频率为 10^6 Hz，适合于测控系统的控制量少，而且对体积和重量的要求较高的场合。但对于区别类似于频率为 3 000 Hz（计数为 666.887）和频率为 2 999 Hz（计数为 666.889）的输出是无法实现的，在这种情况下就需要使用外部计数器。

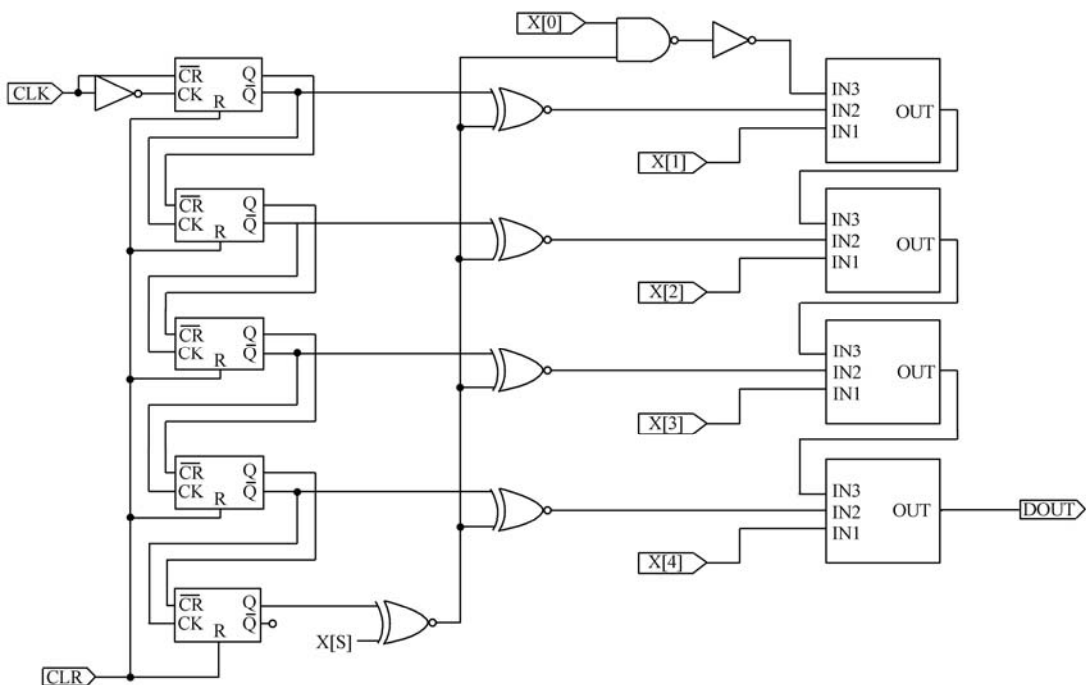


图 6-44 数字 PWM 实现电路

本例选用 Intel 公司的可编程定时/计数器芯片 8254 最为单片机的外部计数器。8254 的主要性能如下。

- 与 Intel 以及大部分公司的微处理器接口兼容；
- 最高计数频率可达 10 MHz；
- 具有状态读回命令；
- 具有 6 种可编程计数器模式；
- 具有 3 个独立的 16 位计数器，可用程序设置成多种工作方式；
- 具有二进制或者十进制两种计数方式；
- 单电源+5 V 供电；
- 可用于实现外部事件计数器、可编程方波频率发生器、分频器、实时时钟以及程控单脉冲发生器等。

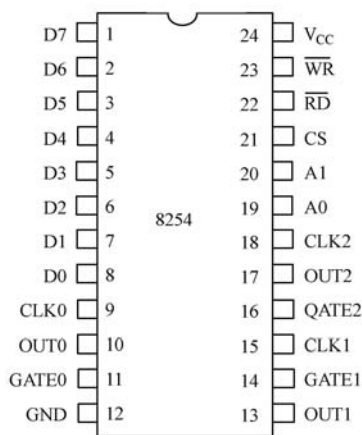


图 6-45 计数/定时芯片 8254 的引脚分布图

(1) 引脚分布及内部结构

可编程计数/定时芯片 8254 的引脚分布如图 6-45 所示。

计数/定时芯片 8254 的引脚功能如表 6-17 所示。

表 6-17 计数/定时芯片 8254 引脚功能说明

引脚名称	引脚数	功 能 说 明
$D_7 \sim D_0$	1~8	双向三态数据线，和系统数据总线相连
CLK0	9	计数器 0 时钟输入端
OUT0	10	计数器 0 输出端
GATE0	11	计数器 0 门输入端
GND	12	电源地
OUT1	13	计数器 1 输出端
GATE1	14	计数器 1 门输入端
CLK1	15	计数器 1 时钟输入端
GATE2	16	计数器 2 门输入端
OUT2	17	计数器 2 输出端
CLK2	18	计数器 1 时钟输入端
A0、A1	19~20	地址线，选择进行读/写操作的 3 个计数器或者控制字寄存器
\overline{CS}	21	片选端
\overline{RD}	22	读控制端
\overline{WR}	23	写控制端
V_{CC}	24	电源，+5 V 供电

计数/定时芯片 8254 的内部时钟示意图如图 6-46 所示。

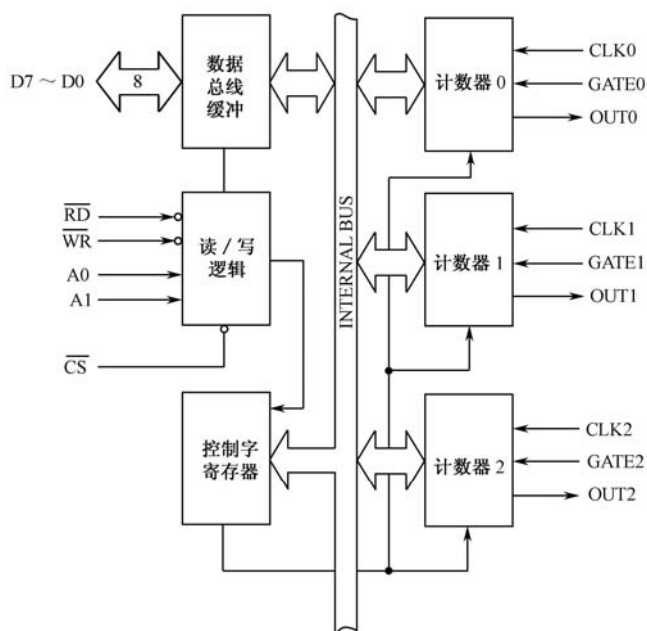


图 6-46 计数/定时芯片 8254 内部时钟示意图

本接口卡的功能组成非常灵活，通过跨接插座的不同连接方式，可以使 8254 的时钟输入端 CLK 与被测现场信号相连，或者与接口卡上基准时钟相连，也可以将二至三级计数器

串联使用。对于 8254 的启停控制端 GATE，同样可以通过跨接插座的选择，使其或者受程序的控制或者设置为外部控制。

(2) 8254 的内部控制字和计数寄存器

计数/定时芯片 8254 内部有 3 个独立的计数器，每个计数器都对应于一个计数寄存器。还有一个控制字寄存器，用于对实现 3 个计数寄存器的配置。

计数/定时芯片 8254 的寄存器的地址是 A1、A0，单片机通过对 A1、A0 的设置，选择对控制字寄存器还是计数寄存器进行操作，A0、A1 对寄存器的选择如表 6-18 所示。

表 6-18 A0、A1 对寄存器的选择

A1	A0	寄 存 器
0	0	计数器 0
0	1	计数器 1
1	0	计数器 2
1	1	控制字寄存器

控制字寄存器的作用是用于选择寄存器的工作方式的，8254 控制字格式如表 6-19 所示。

表 6-19 8254 控制字格式

SC1	SC0	RW1	RW0	M2	M1	M0	BCD
D7	D6	D5	D4	D3	D2	D1	D0

8254 控制字中对应的各位功能如下面几个表所示。

BCD：计数方式选择。设置如表 6-20 所示。

表 6-20 BCD 计数方式选择

BCD	数 码 形 式
0	16 位二进制计数
1	4 位十进制（BCD）码计数

M2、M1、M0 工作方式选择位，设置如表 6-21 所示。

表 6-21 M2、M1、M0 工作方式选择

M2	M1	M0	计数工作方式
0	0	0	方 式 0
0	0	1	方 式 1
×	1	0	方 式 2
×	1	1	方 式 3
1	0	0	方 式 4
1	0	1	方 式 5

RW1、RW0：读/写方式控制字，设置如表 6-22 所示。

表 6-22 RW1、RW0 CPU 读/写操作

RL1	RL0	操 作 类 型
0	0	计数器封锁操作
0	1	读/写计数器低 8 位
1	0	读/写计数器高 8 位
1	1	先读/写低 8 位，后读/写高 8 位

SC1、SC0：选择计数器或读回命令，设置如表 6-23 所示。

表 6-23 SC1、SC0 计数器选择

SC1	SC0	选择计数器
0	0	选择 0#
0	1	选择 1#
1	0	选择 2#
1	1	读回命令

(3) 工作方式

8254 的三个计数器是独立的 16 位减法计数器。计数器的工作方式由工作方式寄存器确定。计数器在编程写入初始值后，在某些方式下计数到 0 后自动预置，计数器连续工作。CPU 访问计数器时，必须先设定工作方式控制字中的 RL1、RL0 位。计数器对 CLK 计数输入端的输入信号进行递减计数。选通信号 GATE 控制计数工作的进行，其功能如表 6-24 所示。

表 6-24 选通信号 GATE 的功能

	低电平或进入低电平	上升边沿	高 电 平
方式 0	禁止计数		允许计数
方式 1		1.初始化和计数 2.下一个时钟后清除输出	
方式 2	1.禁止计数 2.使输出立即变为高电平	1.重新装入计数器 2.启动计数	允许计数
方式 3	1.禁止计数 2.使输出立即变为高电平	初始化和计数	允许计数
方式 4	禁止计数	计数未结束时初始化和计数	允许计数
方式 5		初始化和计数	

8254 的三个计数器按照各工作方式寄存器中控制字的设置进行工作，可以选择的工作方式有 5 种。

这 5 种方式如下：

方式 0：计数结束时中断。编程后自动启动，计数器减 1 计数，计数到终点（减至 0）

后输出高电平，可用于中断请求信号，GATE 为低电平时停止计数，回到高电平后继续递减计数，再次启动要重新装入计数值或重新编程。

方式 1：可编程单脉冲输出。GATE 上升沿进行初始化并开始计数，输出低电平的宽度等于计数时间，单脉冲输出可用 GATE 上升沿多次触发。

方式 2：比率发生器。编程后重复地循环计数，计数到终点时输出一个时钟周期宽度的低电平脉冲，自动初始化后继续计数。用 GATE 的上升沿初始化，并开始计数，GATE 为低电平时停止计数。

方式 3：方波发生器。这种方式是在编程后重复地循环计数，输出波形为方波。如果初始计数值为偶数，每个时钟输入脉冲使计数器减 2，达到计数终点时输出电平改变。如果初始计数值为奇数，则输出高电平时第一个时钟输入脉冲使计数器减 1，随后每个输入脉冲使计数器减 2；输出为低电平时第一个时钟输入脉冲使计数器减 3，随后每个输入脉冲使计数器减 2，到达计数终点时输出电平改变，计数器自动初始化后继续计数。用 GATE 的上升沿初始化并开始计数，GATE 为低电平时停止计数。

方式 4：软件启动选通脉冲输出。编程后自动启动，计数到终点后输出一个时钟周期的低电平脉冲。用 GATE 的上升沿初始化并开始计数，GATE 为低电平时停止计数。

方式 5：硬件启动选通脉冲输出。编程后，等待 GATE 上升沿进行初始化并开始计数，计数到终点后输出一个时钟周期的低电平脉冲。计数器开始计数后不受 GATE 信号电平的影响，这种选通脉冲的输出可用 GATE 的上升沿多次触发。在工作方式控制字中，如果设置计数器锁存操作，则该控制字中工作方式选择位 M1、M0 和计数方式选择位 BCD 无效，即设置锁存操作时不影响计数器的工作方式。计数器锁存操作是在计数器计数过程中，在不影响正在进行的计数操作的条件下，把当前的计数值锁存到寄存器，供 CPU 读取，这时在工作方式控制字中，SC1、SC0 指定要锁存的计数器，RW1=0、RW0=0 表示锁存操作，其余 4 位无效，计数器按原来设定的方式工作。

6.6.3 硬件电路设计

本例将实现 3 路 PWM 信号输出，定时/计数芯片 8254 具有 3 个独立的计数器，只需要选择一片就可以满足要求。硬件电路设计部分主要由单片机控制部分电路、定时/计数芯片 8254 电路以及单片机与定时/计数芯片 8254 的接口电路组成。

单片机控制部分电路原理如图 6-47 所示。

图中所示，单片机部分采用 Atmel 公司的 AT89C52 (U1)，工作时钟为 12 MHz，P0.0~P0.7 口与计数芯片 8254 的数据口 D0~D7 相连，8254 的片选信号线连接在单片机的 P2.0 口，P2.1、P2.2 与 8254 的地址线 A1、A0 相连，8254 的读、写控制引脚分别连接在单片机的 P3.7 口（读）、P3.6 口（写）。

定时/计数芯片 8254 电路原理如图 6-48 所示。

定时/计数芯片 8254 (U2) 由 3 个计数器输出 3 路独立的 PWM 信号，分别为 PWM_1、PWM_2 和 PWM_3，3 个计数器的输入时钟均为 1 MHz，它们的门控制输入均接高电平，确保信号连续输出。

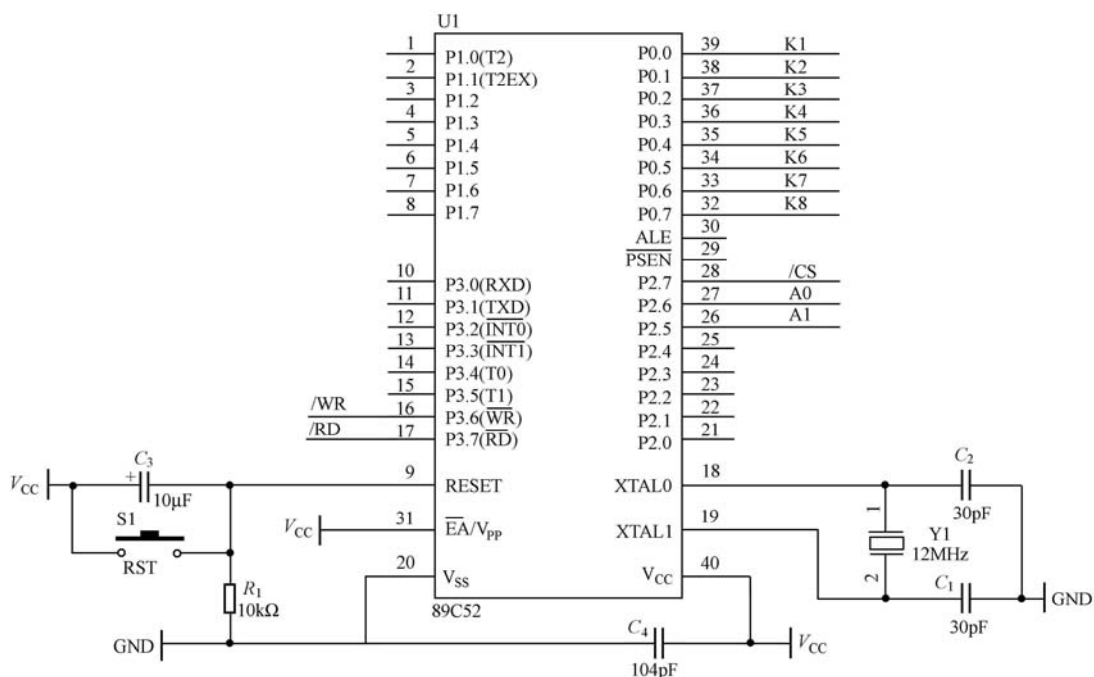


图 6-47 单片机控制部分电路原理图

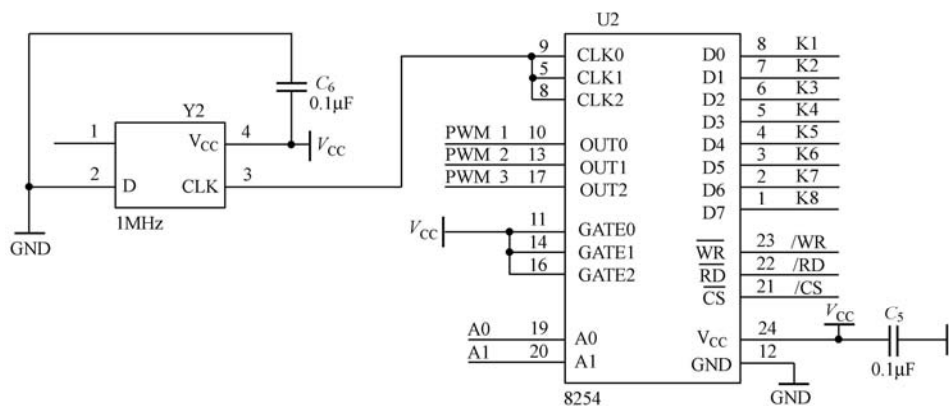


图 6-48 定时/计数芯片 8254 电路原理图

6.6.4 软件设计

单片机主要通过对定时/计数芯片 8254 内部的控制字和计数寄存器的操作来实现控制的。

1. 程序流程

单片机实现 3 路 PWM 信号输出的流程如图 6-49 所示。

2. 程序说明

基于 8254 产生 PWM 信号的程序主要包括三方面内容：一是定义 8254 寄存器的地址，二是控制字的写入，三是数据的写入。

具体代码如下。

```
#include <reg52.h>           //引用标准库的头文件
#include <absacc.h>
#include <stdio.h>

#define COMWORD XBYTE[0x0600] //控制字寄存器地址
#define COUNT0 XBYTE[0x0000]  //计数器 0 寄存器地址
#define COUNT1 XBYTE[0x0200]  //计数器 1 寄存器地址
#define COUNT2 XBYTE[0x0400]  //计数器 2 寄存器地址

void main()
{
    EA = 1;                  //开 CPU 中断
    ET0 = 1;                 //开定时器 0 中断
    TMOD = 0x01;             //定时器 0 工作方式 1
    TH0 = -(20000/256);      //20ms 定时器的计数初值
    TL0 = -(20000%256);

    //选择计数器 0，初值为 0
    COMWORD = 0x30;
    COUNT0 = 0;              //低位字节
    COUNT0 = 0;              //高位字节

    //选择计数器 1，初值为 0
    COMWORD = 0x70;
    COUNT0 = 0;              //低位字节
    COUNT0 = 0;              //高位字节

    //选择计数器 2，初值为 0
    COMWORD = 0xb0;
    COUNT0 = 0;              //低位字节
    COUNT0 = 0;              //高位字节

    TR0 = 1;                 //启动定时器 T0

    while(1){}
```

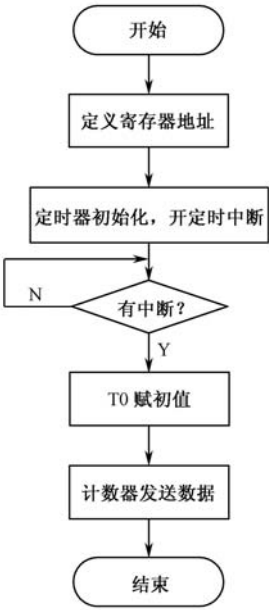


图 6-49 单片机实现 3 路 PWM 信号输出的流程图

```
}
```

```
//定时器 0 中断服务子程序
```

```
void timer0_int() interrupt 1 using 1
```

```
{
```

```
    TR0 = 0;                                //关闭 T0
```

```
    TH0 = -(20000/256);                      //重置 20 ms 定时器的计数初值
```

```
    TL0 = -(20000%256);
```

```
    //发送第 1 路 PWM 信号
```

```
    COMWORD = 0x30;                          //计数 1000 次, 实现 1 ms 高电平
```

```
    COUNT0 = 0xe8;
```

```
    COUNT0 = 0x03;
```

```
    //发送第 2 路 PWM 信号
```

```
    COMWORD = 0x70;                          //计数 2 000 次, 实现 2 ms 高电平
```

```
    COUNT0 = 0xd0;
```

```
    COUNT0 = 0x07;
```

```
    //发送第 3 路 PWM 信号
```

```
    COMWORD = 0xb0;                          //计数 3 000 次, 实现 3 ms 高电平
```

```
    COUNT0 = 0xb8;
```

```
    COUNT0 = 0x0b;
```

```
    TR0 = 1;                                //启动 T0
```

```
}
```

第7章 数据采集系统实例

随着数字技术，特别是信息技术的飞速发展与普及，在现代控制、通信及检测等领域，对信号的处理广泛采用了数字计算机技术。由于系统的实际对象往往都是一些模拟量（如温度、压力、位移、图像等），把这些信号采集、转换为计算机可识别的信号，这些都属于数据采集的应用。

7.1 基于ADC0809 的并行A/D转换

在具有模数转换功能的 C51 单片机出现之前，ADC0809 是与 C51 单片机配合进行 A/D 转换最为典型的、最为常见的芯片。

7.1.1 实例说明

根据 A/D 转换的特点，本例将利用单片机和 A/D 芯片 ADC0809 共同完成 A/D 并行转换的功能。

7.1.2 ADC0809 芯片介绍

ADC0809 是一种 8 路模拟输入逐次比较型 A/D 转换器，由于价格适中，与单片机的接口、软件操作均比较简单，目前在 8 位单片机系统中有着广泛的使用。ADC0809 由 8 路模拟开关、地址锁存与译码器、8 位 A/D 转换器和三态输出锁存缓冲器组成。

ADC0809 基本特性如下：

- 对所有的微处理器都有简单的接口；
- 8 通道的复用器具有逻辑地址；
- 0~+5V 输入范围，具有单个 5V 电源供应；
- 输出满足的 TTL 电压等级的规格；
- 工作温度范围为-40~+85℃；
- 和 MM74C949 功能相当；
- 低功耗，约 15 mW。

ADC0809 的引脚如图 7-1 所示。

1. 结构框图和引脚功能

ADC0809 是 CMOS 单片型逐次逼近式 A/D 转换器，由 8 路模拟开关、地址锁存与译码器、比较器、8 位开关树型 D/A 转换器等组成，内部结构如图 7-2 所示。

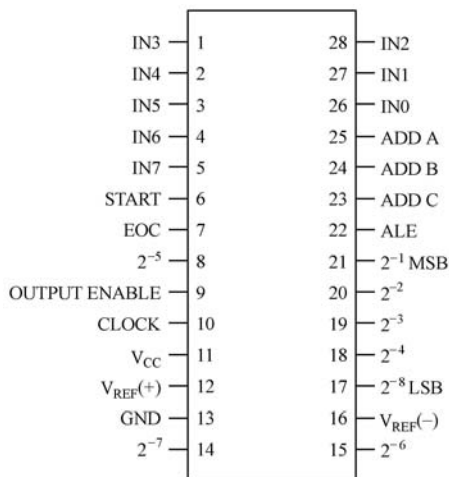


图 7-1 ADC0809 引脚图

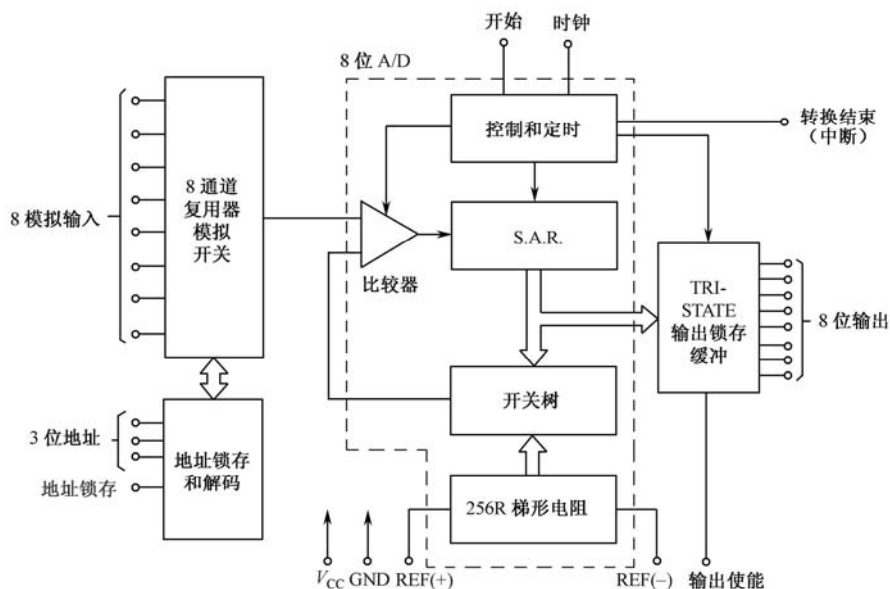


图 7-2 内部结构框图

多路开关可选通 8 个模拟通道，允许 8 路模拟量分时输入，共用一个 A/D 转换器进行转换。地址锁存与译码电路完成对 A、B、C 3 个地址位进行锁存和译码，其译码输出用于通道选择，转换结果通过三态输出锁存器存放、输出，可以直接与系统数据总线相连，表 7-1 为通道选择表。

ADC0809 芯片有 28 个引脚，采用双列直插式封装，它的引脚功能说明如表 7-2 所示。

表 7-1 通道选择表

C	B	A	被选择的通道
0	0	0	IN0
0	0	1	IN1
0	1	0	IN2
0	1	1	IN3
1	0	0	IN4
1	0	1	IN5
1	1	0	IN6
1	1	1	IN7

表 7-2 ADC0809 引脚功能说明

引脚名称	引脚说明
IN0~IN7	8 路模拟量输入引脚
$2^{-1} \sim 2^{-8}$	8 位数字量输出引脚
ADD A、ADD B、ADD C	3 位地址输入线，用于选通 8 路模拟输入中的一路
ALE	地址锁存允许信号输入，高电平有效
START	A/D 转换启动信号输入，高电平有效
EOC	A/D 转换结束信号输出，当 A/D 转换结束时，此端输出一个高电平（转换期间一直为低电平）
CLOCK	时钟脉冲输入引脚。要求时钟频率不高于 640 kHz
$V_{REF}(+)$ 、 $V_{REF}(-)$	基准电压
VCC	电源，单一+5 V
GND	接地
OUTPUT ENABLE	数据输出允许控制引脚，当 OE 端高电平时，控制三态数据输出锁存器向外部输出转换结果数据

2. ADC0809 工作过程

- ① 首先输入 3 位地址，并使 $ALE=1$ ，将地址存入地址锁存器中。
 - ② 此地址经译码选通 8 路模拟输入到比较器。
 - ③ $START$ 上升沿将逐次逼近寄存器复位，下降沿启动 A/D 转换，之后 EOC 输出信号变低，指示转换正在进行。
 - ④ 直到 A/D 转换完成， EOC 变为高电平，指示 A/D 转换结束，结果数据已存入锁存器，这个信号可用做中断申请。
 - ⑤ 当 OE 输入高电平时，输出三态门打开，转换结果的数字量输出到数据总线上。
- ADC0809 的工作时序如图 7-3 所示。

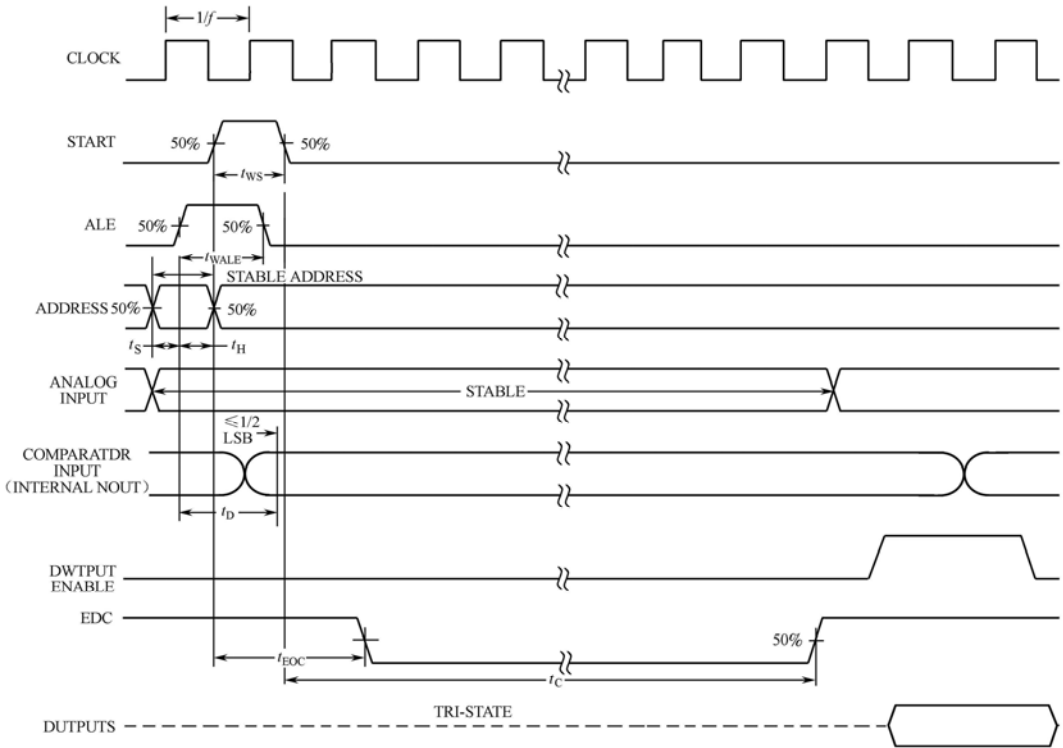


图 7-3 ADC0809 的工作时序示意图

3. A/D转换数据的传送

A/D 转换后得到的数据应及时传送给单片机进行处理。在确认 A/D 转换完成后，才能进行数据的传送。有三种方式可以确认 A/D 转换是否完成。

(1) 定时传送方式

对于 A/D 转换器来说，转换时间作为一项技术指标是已知的和固定的。ADC0809 的转换时间为 $128\ \mu s$ ，相当于 6 MHz 的 C51 单片机共 64 个机器周期。可根据这个指标设计一个延时子程序，A/D 转换启动后调用该子程序，等延迟时间一到，转换肯定已经完成了，就可以进行数据传送了。

(2) 查询方式

A/D 转换芯片具有状态信号，可以表明转换是否完成。可以通过测试 ADC0809 的 EOC

端，利用查询方式，即可知道转换是否完成，并进行数据传送。

(3) 中断方式

将 ADC0809 的状态信号 EOC 作为中断请求信号，以中断方式进行数据传送。

无论使用上述哪种方式，只要确定转换完成，即可通过指令进行数据传送。首先送出口地址并以 \overline{RD} 信号有效时，OE 信号即有效，把转换数据送上数据总线，供单片机接受。

7.1.3 硬件电路设计

从 IN3 通道输入 0~5V 之间的模拟量，并通过 ADC0809 转换成数字量并在数码管上以十进制形式显示出来。ADC0809 的 V_{REF} 接 +5V 电压，硬件电路如图 7-4 所示。

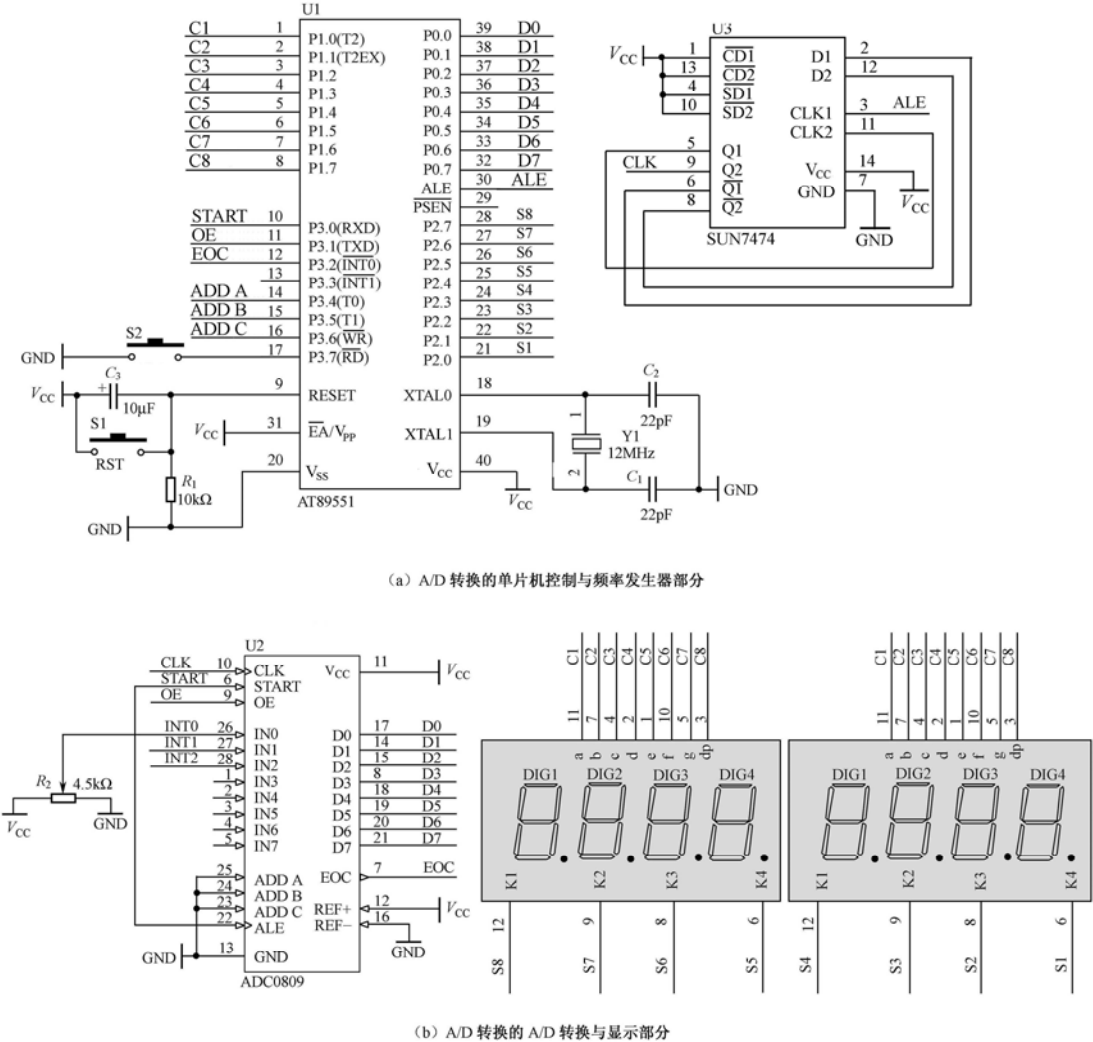


图 7-4 硬件电路图

AT89S51 的 P1.0~P1.7 口用 8 芯排线连接到动态数码显示区域中的 a~g、dp 口上，作为数码管的笔段驱动。P2.0~P2.7 口用 8 芯排线连接到动态数码显示区域中的 S1~S8 口上，作为数码管的位段选择；P0.0~P0.7 口用 8 芯排线连接到 ADC0809 的 D0~D7 口上，A/D 转换完毕的数据输入到单片机的 P0 口。ADC0809 的 VREF 引脚连接到电源模块区域中的 V_{CC} 引脚上。ADC0809 的 A2~A0 引脚用导线连接到 AT89S51 的 P3.4、P3.5、P3.6 口上。ADC0809 的 START 引脚连接到 AT89S51 的 P3.0 口上。ADC0809 的 OE 引脚连接到 AT89S51 的 P3.1 口上。ADC0809 的 EOC 引脚连接到 AT89S51 的 P3.2 口上。

7.1.4 软件设计

软件设计采用查询方式。在进行 A/D 转换时，采用查询 EOC 的标志信号来检测 A/D 转换是否完成，完成后，数据通过 P0 口读入，经过数据处理之后在数码管上显示。在进行 A/D 转换之前，要启动转换的方法：A B C=1 1 0，选择第三通道；START=0，START=1，START=0 产生启动转换的正脉冲信号。

本例程序说明如下：

```
#include <at89x52.h>
unsigned char code disbitcode[]={0xfe,0xfd,0xfb,0xf7,
                                   0xef,0xdf,0xbf,0x7f};

unsigned char code discode[]={0x3f,0x06,0x5b,0x4f,0x66,
                               0x6d,0x7d,0x07,0x7f,0x6f,0x00};

unsigned char disbuf[8]={10,10,10,10,10,0,0,0};
unsigned char discount;

sbit ST=P3^0;
sbit OE=P3^1;
sbit EOC=P3^2;

unsigned char channel=0xbc;    //IN3
unsigned char getdata;
/*主程序*/
void main(void)
{
    TMOD=0x01;
    TH0=(65536-4000)/256;
    TL0=(65536-4000)%256;
    TR0=1;
    ET0=1;
```

```

EA=1;

P3=channel;
while(1)
{
    ST=0;
    ST=1;
    ST=0;
    while(EOC==0);
    OE=1;
    getdata=P0;
    OE=0;
    disbuf[2]=getdata/100;
    getdata=getdata%10;
    disbuf[1]=getdata/10;
    disbuf[0]=getdata%10;

}
}

void t0(void) interrupt 1 using 0           //中断

{

    TH0=(65536-4000)/256;
    TL0=(65536-4000)%256;
    P1=discode[disbuf[discount]];
    P2=disbitcode[discount];
    discount++;
    if(discount==8)
    {
        discount=0;
    }

}

```

7.2 基于TLC549 的串行A/D转换

A/D（模/数）转换器是现代测控中非常重要的转换元件，按转换原理可以分为逐次逼近型、双积分型等；按接口方式可分为串行接口和并行接口；按分辨率又可分为 8 位、12 位、14 位、16 位、18 位等多种类型。在同样的转换分辨率及转换速度的前提下，不同的接

口方式不但影响了电路结构，更重要的是将在高速数据采集的过程中对采样周期产生较大影响。并行 A/D 转换器虽然数据传输速度快，但有引脚多、体积大、占用微处理器接口多的缺点；而串行 A/D 转换器的传输速率目前已经可以做得很高，并且具有体积小、功耗低、占用微处理器接口少的优点。因此，串行 A/D 转换器的应用越来越广泛。

7.2.1 实例说明

单片机已经广泛应用于智能仪表和数据采集领域，促进了测量仪表和测量系统的自动化、智能化。在 A/D 转换系统设计中，实现了将模拟量转换为能被单片机识别的数字量的过程。设计者的主要任务是根据用户对 A/D 转换通道的技术要求，合理地选择通道的结构并按一定的技术准则等因素恰当地选择所需的各种集成电路。

本例主要介绍 8 位 A/D 转换芯片 TLC549 在 C51 单片机系统中的应用，在分析 TLC549 的结构、特点和工作原理的基础上，提供一个硬件连接电路和软件编程。

7.2.2 A/D转换简介

在单片机系统中，最常见的应用是使用 A/D 芯片实现 A/D 转换，下面简单介绍 A/D 转换的原理，以及如何选择合适的 A/D 转换芯片。

1. A/D转换原理

A/D 转换器是将变化连续的模拟信号转换为离散的数字信号，以便于数字系统进行处理、存储、控制和显示。A/D 转换器的形式很多，按 A/D 转换器输入模拟量的极性，可分为单极型和双极型两种；按 A/D 转换器的数字量输出，可分为并行方式、串行方式及串/并行方式；按 A/D 转换器的转换原理，可分为积分型、逐次逼近型和并行转换型。

A/D 转换器的工作原理如图 7-5 所示。

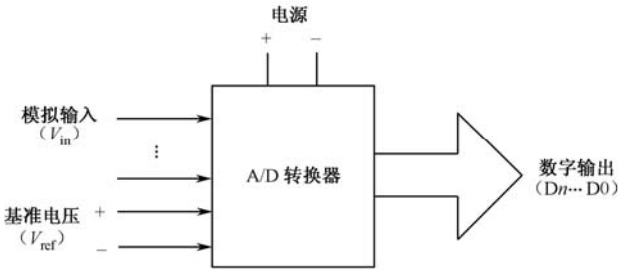


图 7-5 A/D 转换器原理图

由图 7-5 可知，A/D 转换器是一种将模拟信号转换成与之相对应的二进制数的编码器。A/D 转换器有两种输入方式：模拟输入信号 V_{in} 和基准电压 V_{ref} 。

2. 如何选择A/D转换器

A/D 转换器（ADC）是现代电子设备中的关键元件，是控制系统重要的环节，因为它们是模拟信号和数字处理之间的接口，所以常常决定着系统的性能，合理选择 A/D 转换芯片对于确保控制系统的控制精度有着重要的意义，在选择过程中需要考虑器件本身的性能和

具体的应用要求，在实际的选型中应考虑以下方面的因素。

(1) A/D 转换器的分辨率

A/D 转换器的分辨率指数字量变化一个最小量时模拟信号的变化量，定义为满刻度与 $2n$ 的比值。分辨率又称为精度，通常以数字信号的位数来表示。

根据模数转换器位数的不同，可以把模数转换器分为以下几类：

- ① 13 位以上的称为高分辨率 A/D 转换器；
- ② 9~12 位的称为中分辨率 A/D 转换器；
- ③ 把 8 位以下的 A/D 转换器归为低分辨率 A/D 转换器。

A/D 转换器的位数还决定着量化误差的大小，10 位以下的 A/D 产生的误差就比较大了，但选择 10 或 11 位就能满足多数场合的应用。

模拟信号转化为数字信号主要分为两个步骤：首先模拟信号先经过测量装置，然后再经过 A/D 转换器转换后才进行处理。因此，总的误差是由测量误差和量化误差共同构成的。A/D 转换器的精度应与测量装置的精度相匹配。

测量装置的精度与系统中所测量控制的信号范围有关，精度值不小于 $0.1\% \sim 0.5\%$ ，所以选取 A/D 转换器的精度值应在 $0.05\% \sim 0.1\%$ ，换算成相应的二进制码为 10~11 位，加上符号位，为 11~12 位。当对模/数转换器的位数有更高的要求时，可以采用双精度的转换方法来计算 A/D 转换器的位数，转换器位数应该比总精度要求的最低分辨率高一位。

由以上的叙述中可知，选择 A/D 转换器主要应该考虑以下两方面的问题：一方面要求量化误差在总误差中所占的比重要小，使它不显著地扩大测量误差；另一方面必须根据目前测量装置的精度水平，合适地选择 A/D 转换器的位数。

(2) A/D 转换器的转换速率

转换速率是指完成一次从模拟量转换到数字量的 A/D 转换所需的时间的倒数。转换时间的单位是 s，转换速率单位是 ks/s 和 Ms/s，表示每秒采样千/百万次。

在确定 A/D 转换器的转换速率时，应考虑系统的采样速率。采样速率必须小于或等于转换速率，因此有人习惯上将转换速率在数值上等同于采样速率也是可以接受的。例如，如果 A/D 的转换时间为 $100 \mu\text{s}$ 时，则其转换速率为 10 kHz。根据采样定理和实际需要，一个周期的波形需采 10 个样点，那么这样的 A/D 转换器最高也只有处理频率为 1 kHz 的模拟信号。减小转换时间，就可以提高信号采样频率。

(3) 采样/保持器

对于大多数频率较高的模拟信号都要加采样/保持器，但是以下两种情况可以不加采样/保持器：

- ① 采样直流或者变化非常缓慢的模拟信号时，不需要加采样/保持器；
- ② 采样高速 A/D 时，信号频率不高，A/D 转换器的转换时间短，也可不用采样/保持器。

(4) A/D 转换器量程

A/D 转换时需要两种极性：双极性和单极性。输入信号最小值有从零开始的，也有从非零开始的。不同模/数转换器的引脚量程不同，有些 A/D 转换器还提供了不同量程的引脚，只有了解这些引脚的功能，才能正确地应用不同的模/数转换器，才能保证转换精度的准确性。

影响 A/D 转换器量程的因素主要有以下几个方面：量程变换和双极性偏置；双基准电压；A/D 转换器内部比较器输入端的正确使用。

(5) 满刻度误差

满刻度误差是指满度输出时对应的输入信号与理想输入信号值之差。

(6) 线性度

线性度是指实际转换器的转移函数与理想直线的最大偏移。

在进行电路设计时，面对多种多样的 AD 转换芯片，除了要参考以上的诸多因素外，还要考虑综合设计的诸项因素，系统技术指标、成本、功耗、安装等。

3. A/D 转换常见问题

在使用 A/D 转换器的时候，会产生很多问题，下面对如何产生这些问题以及对某些问题的解决方法进行介绍。

(1) A/D 转换器的问题

① A/D 转换器的参考电压值：是 A/D 转换器最大转换电压值，参考电压不得大于器件的电源电压。一般 V_{REF-} 引脚接模拟地； $REF+$ 引脚接参考电压，如果用内置参考电压则该引脚悬空。

② 单片机的电源电压 V_{DD} ：在 A/D 转换中，造成 A/D 转换误差的主要原因是 V_{DD} 电压，如果想要 A/D 转换误差减少到很小，可以调高 V_{DD} 电压精度，一般情况下， V_{DD} 电压精度调节在 0.5%，实际的 A/D 转换误差就可以小于 1%。

③ RC 滤波电路的纹波：在 R_1 、 C_1 取值不当的情况下，U1 处的电压纹波较大，并且延时时间不够，会使 A/D 转换产生误差，因此 R_1 、 C_1 取值不能太小，但太大又会影响 A/D 转换速度，所以说要适当地选择 R_1 、 C_1 的值。在纹波合理的情况下，通过软件也可以消除转换误差。

(2) A/D 转换器的解决方法

① 转换大于参考电压信号的方法：转换大于参考电压信号的通常方法是将信号衰减，将信号的最大值衰减至 A/D 的参考电压值，转换后的数值乘于衰减系数就等于实际的信号值。实际运用中还要考虑电路的其他因数，加上一些修正值。

② 转换正负电压的方法：转换正负电压有多种方法，常用的方法是将信号的正负用运算放大器变换至 $0 \sim +V_{DD}$ ，如信号是对称，在信号输入为 0 时，调运算放大器的输出至 $\frac{1}{2} V_{REF}$ ，使 A/D 转换器输出数值为中点值，测量后要进行换算，如果 A/D 转换器输出数值大于中点值，则输入的信号为正，反之为负；信号的数值等于 A/D 转换器输出数值的中点值。上述的方法会使 A/D 转换器的分辨率降低一半。

③ 对于比较器及 RC 滤波电路的纹波导致的误差，在软件中可通过上、下检测法进行消除，即先将 PWM 的占空比由小到大变化，使电压由低往高逐渐变化，在比较器输出端变化时记录其 A/D 转换值，再将 PWM 的占空比由大到小变化，使电压由高到低变化，在比较器输出端变化时记录其 A/D 转换值，将两次的 A/D 转换值进行平均，可有效地消除这两种误差。

7.2.3 TLC549 芯片介绍

本例采用的单片机是常用的 Atmel 公司 AT89C52 单片机，通过它实现对 A/D 芯片的片选控制，产生满足时序要求的输入/输出时钟，完成对整个 A/D 转换过程的控制。

1. 芯片性能

TLC549 是 TI 公司推出的低功耗、易使用的 A/D 转换器，是以 8 位开关电容逐次逼近 A/D 转换器为基础而构造的 CMOS A/D 转换器。它设计成能通过三态数据输出和模拟输入与微处理器或外围设备串行接口，仅用输入/输出时钟（I/O CLOCK）和芯片选择（CS）输入作为数据控制。

TLC549 的 I/O CLOCK 输入频率最高可达 1.1 MHz，提供了片内系统时钟，它通常工作在 4 MHz 并且不需要外部元件，片内系统时钟使内部器件的操作独立于串行输入/输出的时序并允许 TLC549 像许多软件和硬件所要求的那样工作，I/O CLOCK 和内部系统时钟一起可以实现高速数据传送以及对于 TLC549 为 40 000 次转换/秒的转换速度。

TLC549 的其他特点包括通用控制逻辑，可自动工作或在微处理器控制下工作的片内采样-保持电路，具有差分高阻抗基准电压输入端，易于实现比率转换的高速转换器、定标以及与逻辑和电源噪声隔离的电路。整个开关电容逐次逼近转换器电路的设计允许在小于 17 μ s 的时间内以最大总误差为 ± 0.5 LSB 最低有效位的精度实现转换。TLC549C 的工作温度范围为 0~70℃，TLC549I 的工作温度范围为 -40~85℃。

综上所述，TLC549 主要性能可以总结为以下几点：

- (1) 采用三线串行方式与微处理器接口；
- (2) 8 位分辨率 A/D 转换器，总的非误差 $\leq \pm 0.5$ LSB；
- (3) 片内提供 4 MHz 内部系统时钟，并与操作控制用的外部 I/O CLOCK 相互独立；
- (4) 有片内采样保持电路，转换时间 $\leq 17 \mu$ s；
- (5) 存取与转换时间转换速率达 40 000 次/s；
- (6) 差分高阻抗基准电压输入，其范围是：1 V \leq 差分基准电压 $\leq V_{CC} + 0.2$ V；
- (7) 宽电源范围：3~6.5 V；
- (8) 低功耗，当片选信号 \overline{CS} 为低，芯片选中处于工作状态时，功耗 ≤ 15 mW；
- (9) 采用 CMOS 技术。

A/D 转换芯片采用 TI 公司的 TLC549，其内部结构和引脚分布分别如图 7-6 和图 7-7 所示。

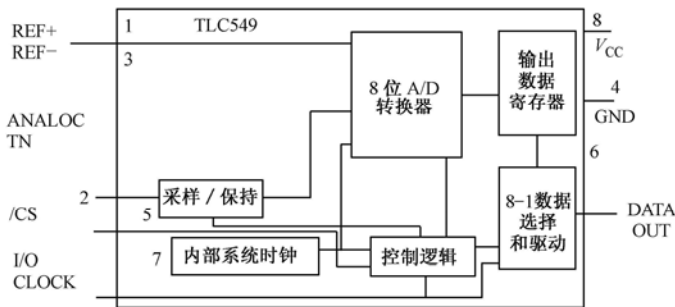


图 7-6 内部结构框图

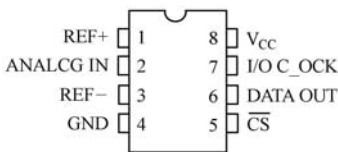


图 7-7 TLC549 引脚分布

各引脚功能说明如表 7-3 所示。

表 7-3 引脚功能说明

引脚名称	说 明
REF+	正基准电压输入引脚, $2.5\text{ V} \leq \text{REF+} \leq V_{\text{CC}}+0.1$
ANALOG IN	模拟量串行输入引脚, $0 \leq \text{ANALOG IN} \leq V_{\text{CC}}$, 当 $\text{ANALOG IN} \geq \text{REF+}$ 电压时, 转换结果为全 1(FFH), $\text{ANALOG IN} \leq \text{REF-}$ 电压时, 转换结果为全 0(00H)
REF-	负基准电压输入引脚, $-0.1\text{ V} \leq \text{REF-} \leq 2.5\text{ V}$, 要求 $\text{REF+} - \text{REF-} \geq 1\text{ V}$
GND	接地端
$\overline{\text{CS}}$	芯片选择输入引脚, 低电平有效, 要求输入高电平 $V_{\text{IN}} \geq 2\text{ V}$, 输入低电平 $V_{\text{IN}} \leq 0.8\text{ V}$
DATA OUT	转换结果数据串行输出引脚, 与 TTL 电平兼容, 输出时高位在前, 低位在后
I/O CLOCK	外接输入/输出时钟输入引脚, 等同于同步芯片的输入/输出操作, 无需与芯片内部系统时钟同步
VCC	系统电源, $3\text{ V} \leq V_{\text{CC}} \leq 6\text{ V}$

由以上可以看出, TCL549 可以使用差分基准电压, 这是该芯片的重要特性, 利用这个特性 TCL549 可能测量到的最小量值达 $1\,000\text{ mV}/256$, 也就是说 $0\sim 1\text{ V}$ 信号不经放大也可以得到 8 位的分辨率, 因此可以简化电路、节省成本。

2. 工作原理

TCL549 芯片包含内部系统时钟、采样/保持电路、8 位 A/D 转换电路、输出数据寄存器以及控制逻辑电路, 它采用 $\overline{\text{CS}}$ 、I/O CLOCK 和 DATA OUT 三根线实现与微控制器 MCU 或微处理器 CPU 进行串行通信, 其中 $\overline{\text{CS}}$ 和 I/O CLOCK 作为输入控制, 芯片选择端 $\overline{\text{CS}}$ 低电平有效, 当 $\overline{\text{CS}}$ 为高电平时 I/O CLOCK 输入被禁止, 且 DATA OUT 输出处于高阻状态, 其工作时序如图 7-8 所示。

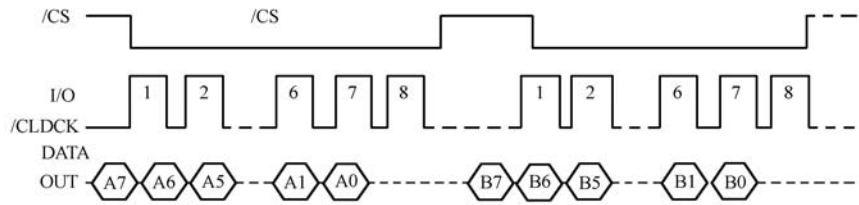


图 7-8 TCL549 工作时序示意图

当 $\overline{\text{CS}}$ 由高电平变为低电平后, TCL549 芯片被选中, 内部电路在测得 $\overline{\text{CS}}$ 下降沿后, 再等待两个内部时钟上升沿和一个下降沿, 确认变化后, 将前一次转换结果的最高有效位 MSB (A7) 从 DATA OUT 引脚输出。

接下来要求自 I/O CLOCK 引脚输入 8 个外部时钟信号, 前 7 个 I/O CLOCK 信号的作用, 是配合 TLC 549 输出前次转换结果的 A6~A0 共 7 位, 并为本次转换做准备。

在第 4 个 I/O CLOCK 信号由高至低的跳变之后, 片内采样/保持电路对输入模拟量采样开始, 第 8 个 I/O CLOCK 信号的下降沿使片内采样/保持电路进入保持状态并开始启动 A/D 转换。转换时间为 36 个系统时钟周期, 最大为 $17\text{ }\mu\text{s}$, 直到 A/D 转换完成前的这段时间

内，TCL549 的控制逻辑要求 \overline{CS} 保持高电平，或者 I/O CLOCK 时钟端保持 36 个系统时钟周期的低电平。

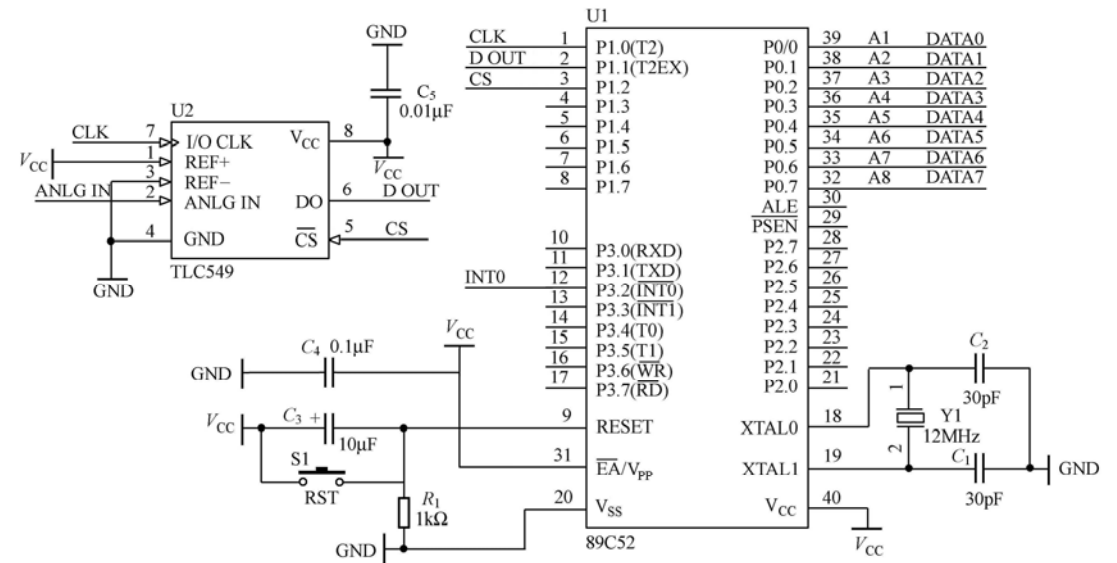
由此可见，在自 TCL549 芯片的 I/O CLOCK 引脚输入 8 个外部时钟信号期间需要完成以下工作：读入前次 A/D 转换结果；对本次转换的输入模拟信号采样并保持；启动本次 A/D 转换开始。

如果想在特定的时刻采样模拟信号，应使第 8 个 I/O CLOCK 时钟的下降沿与该时刻对应，因为 TCL549 芯片在第 4 个 I/O CLOCK 时钟下降沿开始采样，却在第 8 个 I/O CLOCK 时钟的下降沿开始保存。

当 \overline{CS} 由低电平变为高电平时，数据输出引脚（DATA OUT）处于高阻态，此时 I/O CLOCK 不起作用。 \overline{CS} 的这种控制方式适合在同时使用多片 TCL549 芯片时共用 I/O CLOCK，用以减少多路 A/D 芯片并用时的 I/O 控制端口。

7.2.4 硬件电路设计

单片机和 A/D 转换器之间的接口设计是硬件电路设计的关键。TLC549 芯片可方便地与具有串行外围接口(SPI)的单片机或微处理器配合使用，也可与 C51 通用单片机连接使用。串行 A/D 转换电路单片机部分原理图如图 7-9 所示。



此在用做标准 I/O 口使用时，应外接上拉电阻。电路中正基准电压 REF_+ 接+5 V，负基准电压 REF_- 接地，因此测量范围是 0~5 V，对应输出数字量是 0~255。

7.2.5 软件设计

软件的设计思想是：在主程序中，通过置 \overline{CS} （P1.2 口）端为低选中芯片，调用读入 8 位转换结果子程序，开始启动 A/D 转换，置 \overline{CS} （P2.2 口）端为高并延时 17 μs 等待 A/D 转换结束。当 A/D 转换结束后，置 \overline{CS} （P1.2 口）端为低电平，再次调用读入 8 位转换结果子程序，则可读入本次 A/D 转换结果。在读入 8 位转换结果子程序中，通过程序语句读入前次转换结果的最高位，接着用循环程序在 P1.1 口上发出 7 个 I/O CLOCK 脉冲，同时读入转换结果的其余 7 位，最后通过程序语句在 P1.1 口发出第 8 个 I/O CLOCK 脉冲，从而启动本次 A/D 转换。

1. 程序流程

单片机实现串行 A/D 转换程序的流程如图 7-10 所示。

2. 程序说明

```
#include <reg52.h> // 引用标准库的头文件
#include <intrins.h>

#define uchar unsigned char

uchar DResult; // 存放转换后数据
sbit IOCLK = P1^0; // 输入输出时钟
sbit DATAOUT = P1^1; // 数据输出
sbit CS = P1^2; // 片选信号

void main()
{
```

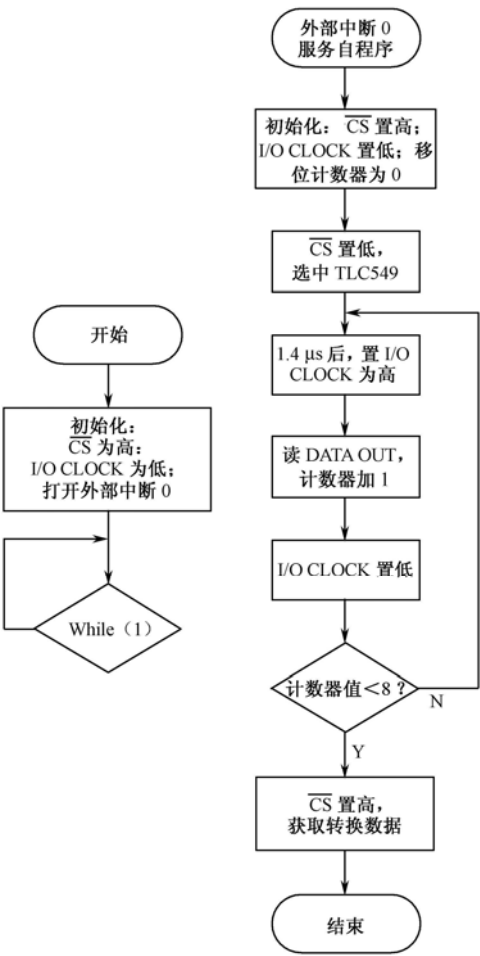


图 7-10 单片机实现串行 A/D 转换程序流程

```

    EA = 1;
    EX0 = 1;                //开中断
    while(1);
}

void int0svr(void) interrupt 0 using 1
{
    uchar count;
    uchar tmp;
    uchar t;

    EX0 = 0;                // 关中断

    tmp = 0;
    CS = 1;                 // 片选无效
    IOCLK = 0;

    CS = 0;                 // 片选有效
    _nop_();                // 空指令延时

    for (count=0;count<8;count++)
    {
        IOCLK = 1;
        if (DATAOUT)
            tmp++;
        tmp = tmp<<1;
        IOCLK = 0;
    }

    CS = 1;                 // 片选无效

    DResult = tmp;

    for(t=0;t<3;t++)        // 延时
        _nop_();

    EX0 = 1;                // 开中断
}

```

7.3 基于MAX532 的串行D/A转换

串行 D/A 转换器由于接口电路简单、易于远程操作以及体积小、功耗低等优点而广泛应用于便携式设备或分布式控制系统中。

7.3.1 实例说明

在制作单片机系统时，通常会需要将处理器的结果转换成模拟量，从而对某些对象的控制就会更简单。

本例是在 C51 单片机系统中基于 MAX532 实现并行 D/A 转换的典型实例。

7.3.2 D/A转换

简单地说，D/A 转换就是将离散的数字量转换为连续变化的模拟量，实现该功能的电路称为数/模转换电路，通常称为 D/A 转换器或 DAC（digital analog converter）。

1. D/A转换器的工作原理

D/A 转换器是将一组输入的二进制数转换成相应数量的模拟电压或电流输出的电路，工作原理是将每一位二进制数按其权的大小转换成相应的模拟量，然后将代表各位的模拟量相加，使所得的总模拟量与数字量成正比。数/模转换器实质上是由二进制数字量控制模拟电子开关，再由模拟电子开关控制电阻网络与运算放大器组成的模拟加法运算电路。

2. D/A转换器常用的解码网络

D/A 转换器的具体电路有多种形式，其中解码网络是普通采用的形式，解码网络的主要形式有二进制权电阻网络和 T 型电阻网络 2 种。

图 7-11 为 4 位二进制权电阻网络的电路图，运算放大器的输出电压为

$$u_o = -R_f I_f = -\frac{R_f U_R}{2^3 R} (d_3 \cdot 2^3 + d_2 \cdot 2^2 + d_1 \cdot 2^1 + d_0 \cdot 2^0)$$

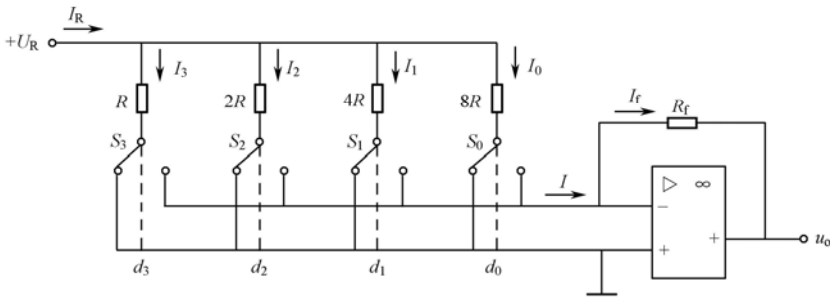


图 7-11 二进制权电阻网络

图 7-12 为 T 型电阻网络的电路图，当 $R_f = 3R$ 时，其输出电压 u_o 为

$$u_o = -\frac{U_R}{2^4} (d_3 \cdot 2^3 + d_2 \cdot 2^2 + d_1 \cdot 2^1 + d_0 \cdot 2^0)$$

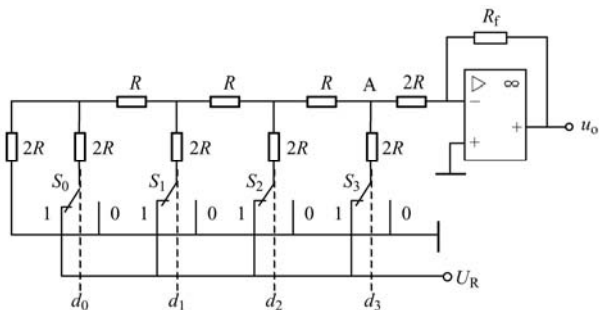


图 7-12 T 型电阻网络

7.3.3 MAX532 芯片介绍

MAX532 是由 MAXIM 公司生产的一款带有输出放大器的双路串行 12 位电压输出的三线串行输入 D/A（数模转换器），供电范围在 12~15 V，所有逻辑输出端口与 TTL 和 CMOS 兼容。

1. MAX532 的内部结构

MAX532 的内部结构如图 7-13 所示。

2. MAX532 的引脚

MAX532 的引脚如图 7-14 所示。

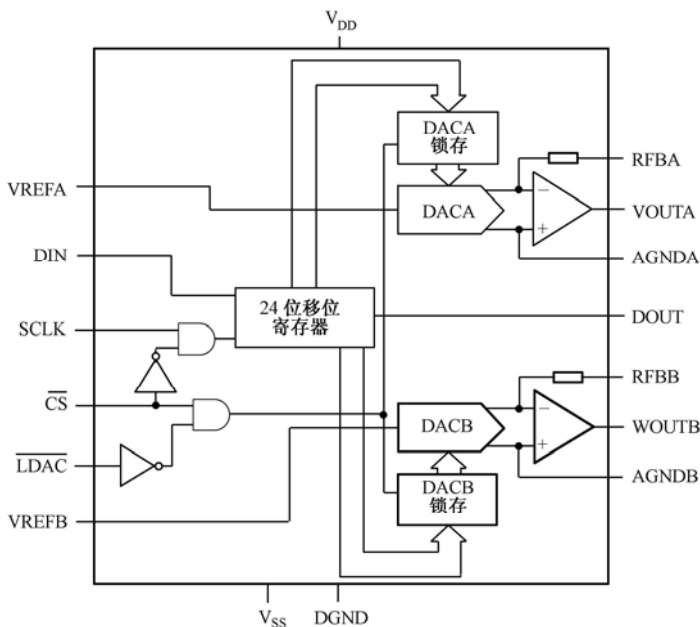


图 7-13 MAX532 的内部结构图

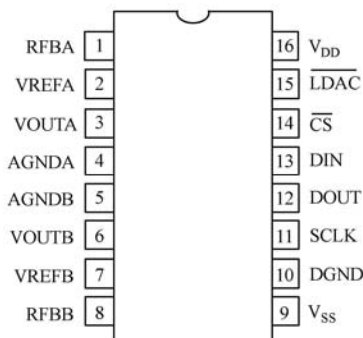


图 7-14 MAX532 的引脚图

各引脚的功能说明如表 7-4 所示。

表 7-4 MAX532 的引脚说明

引 脚 名 称	说 明
V _{DD} , V _{SS}	正, 负电源
DGND	数字地
$\overline{\text{CS}}$	芯选引脚, 低电平有效
$\overline{\text{LDAC}}$	异步加载 D/A 输入, 低电平有效
DIN, DOUT	串行数据的输入, 输出
SCLK	串行时钟输入
VOUTB, VREFB, RFBB	DACB 的模拟地, 电压输出, 参考电压输入
VOUTA, VREFA, RFBA	DACA 的模拟地, 电压输出, 参考电压输入
AGNDA, AGNDB	DACA, DACB 的模拟地

3. MAX532 的时序图

MAX532 有三线和四线两种接口方式。

(1) 三线接口方式

使用 $\overline{\text{CS}}$, DIN, SCLK 引脚, $\overline{\text{LDAC}}$ 始终接低电平, D/A 随着片选信号 $\overline{\text{CS}}$ 升高电平时同步更新, 其时序如图 7-15 所示。

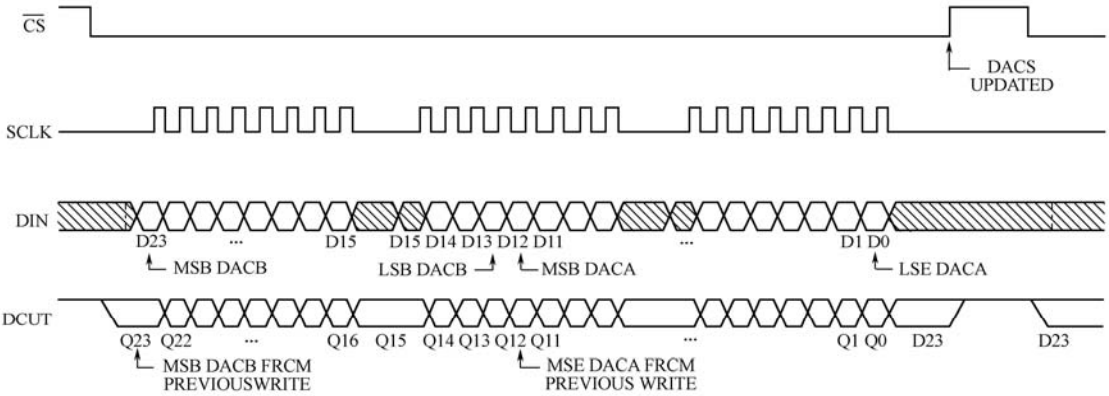


图 7-15 三线接口方式时序图

(2) 四线接口方式

多用于串行器件连接到统一的数据线。当 $\overline{\text{LDAC}}$ 变为低电平时, 所有串行器件同步更新。其时序图如图 7-16 所示。

控制器通过串行输入三字节的数据来给出两通道 12 位 DAC 寄存器 (DACA、DACB) 的数字值。串行数据先被锁定至 DACB 的数据寄存器, 然后再至 DACA, MSB 数据位在前。当片选引脚 $\overline{\text{CS}}$ 位低电平时, 数据在时钟信号 SCLK 的上升沿被锁入; 当片选引脚 $\overline{\text{CS}}$ 为高电平时, 数据不能被读至 DIN, 它能够使 DOUT 为高阻抗状态。SCLK 的频率可达 6.25 MHz。

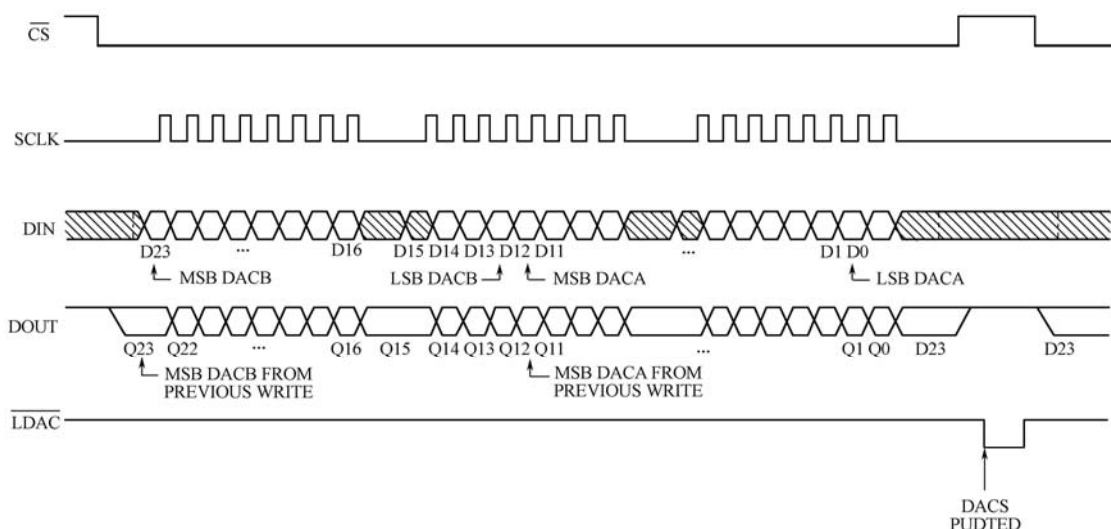


图 7-16 四线接口方式时序图

4. MAX532 的主要性能

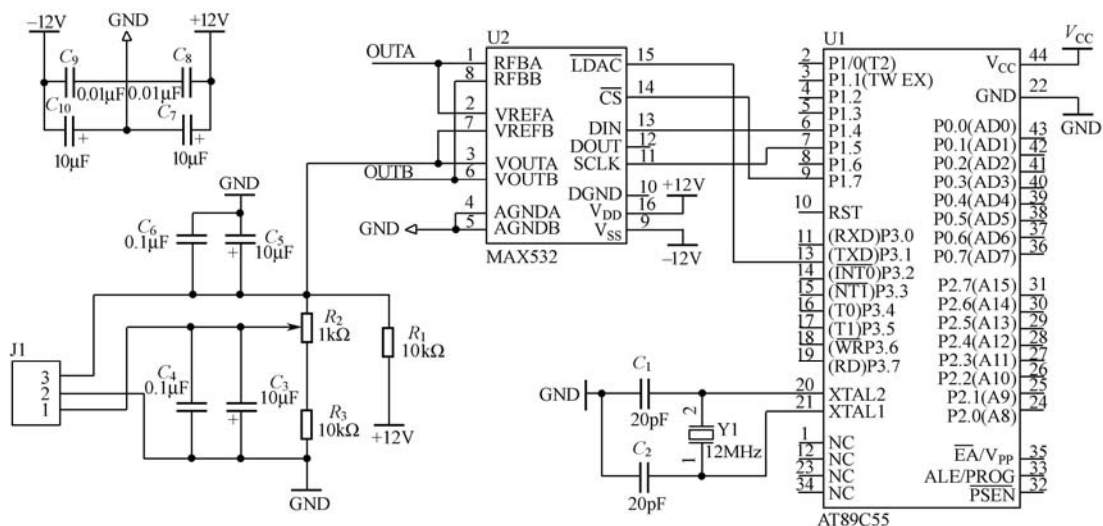
数模转换芯片 MAX532 的主要性能如下：

- 电压输出范围：-12~+12 V；
- 电流输出范围：-10~+10 mA；
- 工作电压范围：±12~±15 V；
- 在温度范围内其变化趋势是单调的；
- 双路带有放大器的 12 位 D/A 转换器；
- 高速 6 MHz 三线接口；
- 与 SPI、QSPI 和 MICROWITE 接口标准兼容；
- 整体非线性程度低，小于±0.5 LSB；
- 内部上电复位。

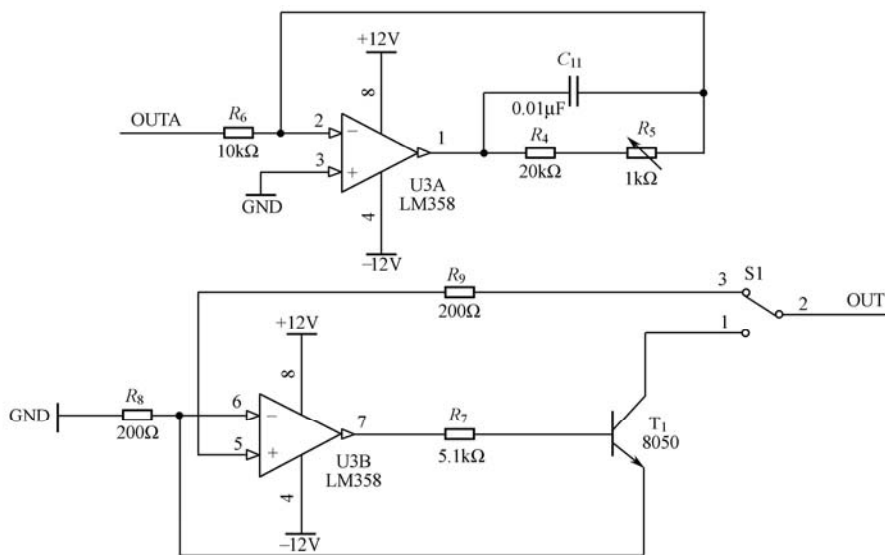
7.3.4 硬件原理图的设计

MAX532 与 C51 单片机的接口电路如图 7-17 所示。

LM358 是双运算放大器，用于将来自 OUTA 输出的电压反向并放大一倍，431 (U4) 是可调并联稳压器用于提供 MAX532 的参考电压；S1 用于选择电压输出或电流输出；P1.7 口作为片选信号，P1.5 口提供了 MAX532 的时钟信号；P1.4 口时数据输入信号，P3.1 口接到 MAX532 的 $\overline{\text{LDAC}}$ 引脚提供同步信号。



(a) 单片机控制部分



(b) LM358 放大部分

图 7-17 MAX532 与 C51 单片机的接口电路图

7.3.5 程序设计

本例的程序设计包含两个子程序，调用时先发送 B 通道数据再发送 A 通道数据，在调用完 DABSent () 后，必须接着调用 DAASent ()。由于 MAX532 在接受数据时是高位在先，所以在模拟串口通信时要先发送高位数据再发送低位数据。

本例的主要子程序代码如下：

```
#include<reg51.h>
#define uchar unsigned char
#define uint unsigned int
```

```

#define AdL8 XBYTE[0xFDFD]    // 高 8 位
#define AdH4 XBYTE[0xFEFF]    // 低 4 位

void DABSent (uchar, uchar);
void DAASend (uchar, uchar);
sbit A8 = P2^0;                // DAC 通道选择引脚 A8
sbit A9 = P2^1;                // DAC 通道选择引脚 A9

void DABSent (uchar DABdatahigh , uchar DABdatalow)    //B 通道数据发送
{
uint i ;
sbit CS=P1^7;                //片选信号
sbit LDAC=P3^1;              //异步加载 DAC 输入作为同步信号
sbit DIN=P1^4;                //数据输入
sbit SCLK=P1^5;              //时钟
uchar temp;
cs=1;
LDAC=1;
CS=0;
temp=DABdatahigh;
for (i=0; i<8; i++)
{
    SCLK=0;
    temp=temp&0x80;
    if (temp==0x80)
        DIN=1;
    else
        DIN=0
    Temp=DABdatahigh;
    Temp=temp<<1;
    DABdatahigh=temp;
    SCLK=1;
}
temp=DABdatalow;
for (i=0; i<4; i++)
{
    SCLK=0;
    Temp=temp&0x80h;
    if (temp=0x80h)
        DIN=1;

```

```

else
    DIN=0;
temp=DABdatalow;
temp=temp<<1;
DABdatalow=temp;
SCLK=1;
}
}

void DAASend (uchar DAAdatashigh, uchar DAAdatalow)
{
uint i ;
sbit CS=P1^7;                //片选信号
sbit LDAC=P3^1;              //异步加载 DAC 输入作为同步信号
sbit DIN=P1^4;               //数据输入
sbit SCLK=P1^5;              //时钟
uchar temp;
LDAC=1;
cs=0;
temp=DABdatashigh;

for (i=0;i<8;i++)
{
    SCLK=0;
    temp=temp&0x80;
    if (temp==0x80)
        DIN=1;
    else
        DIN=0
    temp=DAAdatashigh;
    temp=temp<<1;
    DABdatashigh=temp;
    SCLK=1;
}
temp=DAAdatalow;
for (i=0; i<4; i++)
{
    SCLK=0;
    temp=temp&0x80;
    if (temp==0x80)
        DIN=1;

```

```

else
    DIN=0;
temp=DAAdatalow;
temp=temp<<1;
DAAdatalow=temp;
SCLK=1;
}
cs=1;
LDAC=0;
}
//主函数
void main()
{
    EA = 1;
    EX0 = 1;
    While(1)
    {
        void DABSent (Adl8, Adl4);
        void DAASend (Adl8r, Adl4);
    }
}

```

7.4 基于DS18B20 的数字温度计设计

单片机可以测量温度、湿度等非电信号，并且在测控领域得到了广泛的应用。本例将通过设计一个数字温度计来了解一下单片机是如何进行温度测量的。

7.4.1 实例效果说明

本例将采用 C51 单片机通过 DS18B20 芯片读取室外的温度值，然后利用 MAX7219 芯片进行显示。

7.4.2 DS18B20 芯片介绍

本例采用 MAXIM 公司生产的 DS18B20 芯片，属于单线智能温度传感器，是新一代适配微处理器的智能温度传感器，它广泛用于工业、民用、军事等领域的温度测量及控制仪器、测控系统和大型设备中。

DS18B20 有三种封装形式，本例采用 8 引脚的 SO 封装，它的引脚如图 7-18 所示。各引脚功能说明如表 7-5 所示。

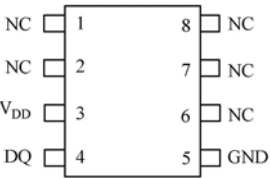


图 7-18 DS18B20 引脚图

表 7-5 DS18B20 引脚功能说明

引 脚 名 称	说 明
V_{DD}	可选的+5 V 电源
DQ	数字输入/输出
GND	电源地
NC	无连接

DS18B20 内部结构如图 7-19 所示。64 位光刻 ROM 是出厂前被光刻好的，它可以看做该 DS18B20 的地址序列号。不同的器件地址序列号不同。DS18B20 内部结构主要由 64 位光刻 ROM、温度传感器、非挥发的温度报警触发器 TH 和 TL 及高速暂存器四部分组成。

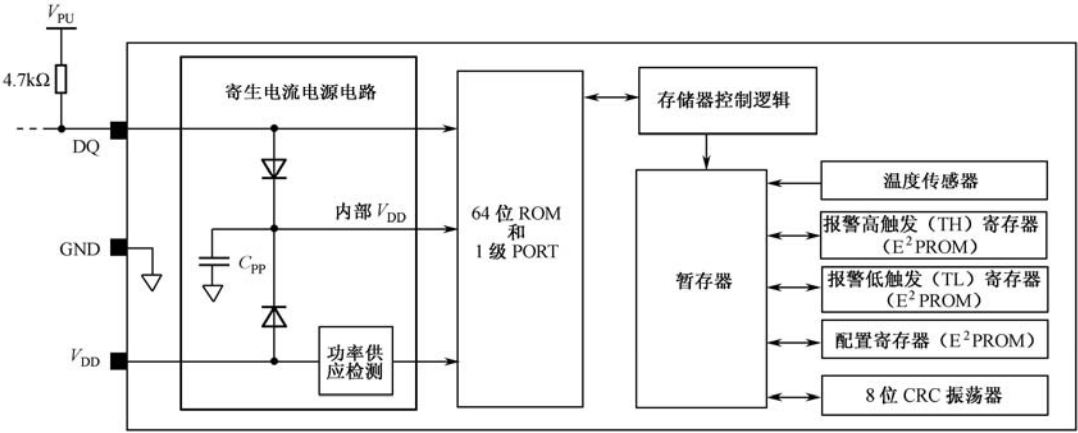


图 7-19 内部结构框图

当信号线为高电平时，内部电容器储存能量由一线通信线路给芯片供电，在低电平期间为芯片供电直至下一个高电平的到来重新充电。DS18B20 的电源也可以从外部 3.0~5.5 V 的电压得到。

DS18B20 的特点如下：

- 适应电压范围更宽，电压范围为 3.0~5.5 V，寄生电源方式下可由数据线提供；
- 独特的单线接口方式，在与微处理器连接时仅需要一条口线，即可实现微处理器与 DS18B20 的双向通信；
- 支持多点组网功能，多个 DS18B20 可以并联在唯一的三线上，实现组网多点测温；
- 不需要任何外围元件，全部传感元件及转换电路集成在形如一只三极管的集成电路内；
- 温度范围为-55~+125 ℃，在-10~+85 ℃时，精度为±0.5 ℃。
- 可编程的分辨率为 9~12 位，对应的可分辨温度分别为 0.5 ℃、0.25 ℃、0.125 ℃和 0.0625 ℃，可实现高精度测温。
- 在 9 位分辨率时，最多在 93.75 ms 内把温度转换为数字，12 位分辨率时最多在 750 ms 内把温度值转换为数字，速度较快。
- 测量结果直接输出数字温度信号，以一线总线串行传送给 CPU，同时可传送 CRC 校

验码，具有极强的抗干扰纠错能力。

- 负压特性：电源极性接反时，芯片不会因发热而烧毁，但不能正常工作。

1. DS18B20 的指令功能

DS18B20 采用的是一线通信接口，而一线通信接口首先必须完成 ROM 的设定，否则记忆和控制功能将无法使用。

首先提供以下指令功能之一：

- 读 ROM；
- ROM 匹配；
- 搜索 ROM；
- 跳过 ROM；
- 报警检查。

这些指令操作作用在 64 位光刻 ROM 序列号上，该序列号上没有一个器件。可以在一线上多个器件中选定某一个器件，同时，总线也可以知道总线上挂了多少个什么样的设备。

若这些指令成功地使 DS18B20 完成温度测量，则数据存储在 DS18B20 的存储器中。温度报警触发器 TH 和 TL 都有 1 B E²PROM 数据。如果 DS18B20 不使用报警检查指令，这些寄存器可作为一般的用户记忆用途。在片上还载有配置字节来解决温度到数字转换。利用一个记忆功能的指令来完成写 TH 和 TL 触发器及配置字节。通过缓存器读寄存器，所有的数据读、写都是从最低位开始。

2. DS18B20 读出数据的计算处理方法

从 DS18B20 读取的二进制数值必须先转换成十进制数值后，才能用于字符的显示。DS18B20 的转换精度可选为 9~12 位，在采用 12 位转换精度时，温度寄存器里的值以 0.0625 为步进，即温度值为温度寄存器里的二进制值乘以 0.0625，得到实际的十进制温度值。十进制值和二进制值及十六进制值之间的关系如表 7-6 所示。

表 7-6 DS18B20 温度与测得值对应表

温度/℃	二 进 制 值	十六进制值
+125	0000 0111 1101 0000	07D0H
+85	0000 0101 0101 0000	0550H
+25.0625	0000 0001 1001 0001	0191H
+10.125	0000 0000 1010 0010	00A2H
+0.5	0000 0000 0000 1000	0008H
0	0000 0000 0000 0000	0000H
-0.5	1111 1111 1111 1000	FFF8H
-10.125	1111 1111 0101 1110	FF5EH
-25.0625	1111 1110 0110 1111	FE6FH
-55	1111 1100 1001 0000	FC90H

把二进制高字节的低半字节和低字节的高半字节组成一个字节，这个字节的二进制值转换为十进制值后，就是温度值的百、十、个位值，而剩下的低字节的低半字节转换成十进制后，就是温度值的小数部分。

因为小数部分是半字节，所以二进制值的范围是 0~F，转换成十进制小数值就是 0.0625 的倍数（0~15 倍）。小数部分二进制和十进制的近似对应关系如表 7-7 所示。

表 7-7 小数部分二进制和十进制的近似对应关系表

小数部分 二进制值	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
十进制值	0	1	1	2	3	3	4	4	5	6	6	7	8	8	9	9

3. DS18B20 时序

DS18B20 是单线控制芯片，操作时要有严格的时序概念，因此读、写时序很重要。系统对 DS18B20 的各种操作必须按协议进行。

操作协议为：初始化 DS18B20（发复位脉冲），发 ROM 功能命令，发存储器操作命令，处理数据。

（1）DS18B20 的复位函数时序

复位函数的时序如图 7-20 所示。

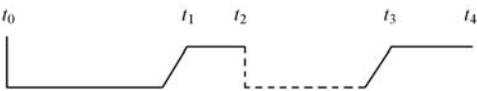


图 7-20 复位函数时序

总线在 t_0 时刻发送一个复位脉冲（最短为 480 ns 低电平信号），在 t_1 时刻释放总线，并进入接收状态，在总线上升沿之后等待 15~60 μs ，在 t_2 时刻发出存在脉冲（低电平持续 60~240 μs ），由图中虚线所示，单片机接收到低电平脉冲，则说明复位成功，否则需要重新进行复位操作。

（2）DS18B20 的写时序

当主机把数据线从逻辑高电平拉到逻辑低电平时，开始写间隙。包括写 1 时间隙和写 0 时间隙。主机生成一个写 1 时间隙，需要把数据线拉到低电平，然后释放。写时间隙开始后的 15 μs 内允许数据线拉到高电平。主机生成一个写 0 时间隙，必须把数据线拉到低电平，并保持 60 μs ，如图 7-21 所示。

（3）DS18B20 的读时序

从 DS18B20 开始读数据时，主机生成一个读时间隙，输出的数据在读时间隙的下降沿出现后 15 μs 内有效。所有的读时间隙必须最少 60 μs ，包括两个读周期至少 1 μs 的恢复时间。所以，主机在读时间隙开始后必须停止把 I/O 口驱动为低电平 15 μs ，用来读取 I/O 口的状态。在读时间隙的结尾，I/O 口被外部上拉电阻拉到高电平，如图 7-22 所示。

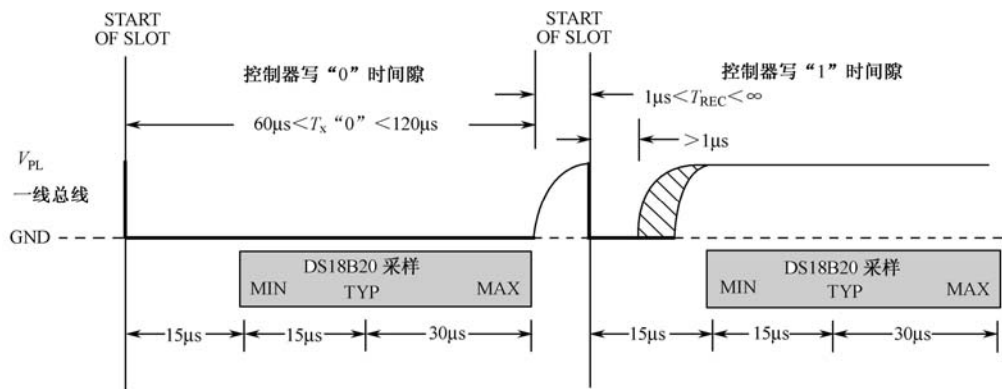


图 7-21 写时序

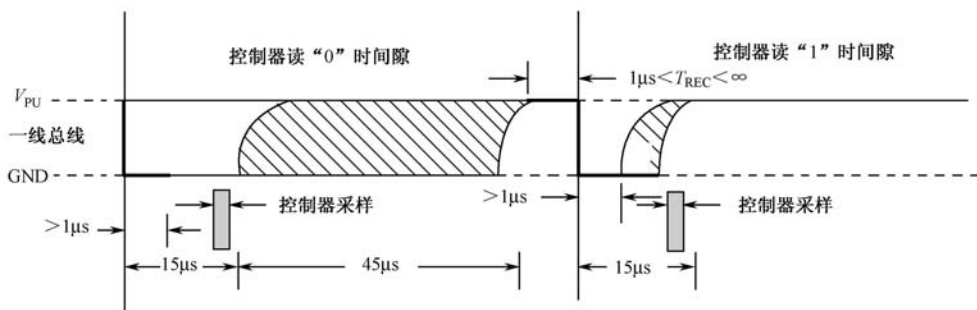


图 7-22 读时序

7.4.3 MAX7219 芯片介绍

本例采用 MAXIM 公司的 AX7219 芯片来进行显示。定义一个 diss 字符数组，把要显示的 8 位温度值所对应的 BCD 码和符号码都存入其中，典型的应用电路如图 7-23 所示。

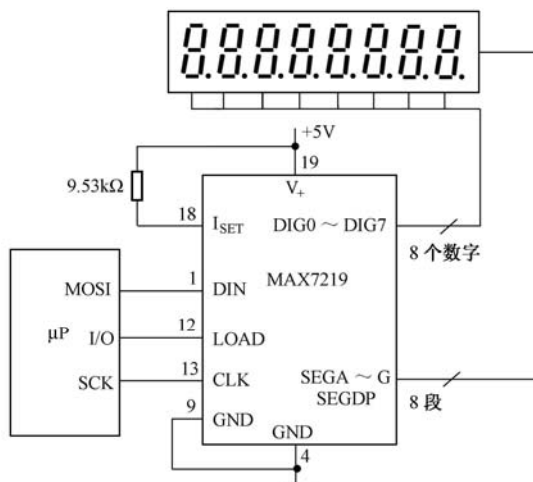


图 7-23 MAX7219 典型应用电路

MAX7219 的引脚如图 7-24 所示。

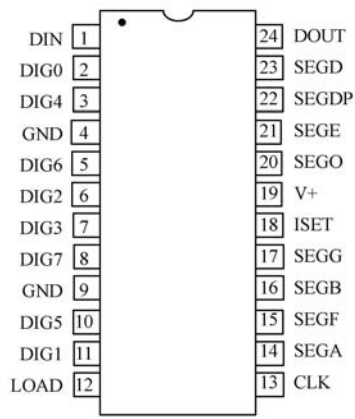


图 7-24 MAX7219 引脚图

各引脚功能说明如表 7-8 所示。

表 7-8 MAX7219 引脚功能说明

引 脚 名 称	说 明
DIN	串行数据输入引脚，在 CLK 的上升沿，数据被移入内部的 16 位移位寄存器中
DIG0~DIG7	输入，8 位数字位驱动线，从共阴极显示器吸收电流
GND	电源地
LOAD	数据装载输入引脚。在 LOAD 上升沿，移位寄存器接受的数据被锁存
CLK	时钟输入引脚。在 CLK 的上升沿，数据被移入内部的 16 位移位寄存器中；在 CLK 的下降沿，数据从 DOUT 时钟输出
SEG A~SEG G	七段驱动和小数点驱动电源电流在显示器上。当一段驱动器关闭时被下拉至接地
ISET	连接电源通过一个电阻（REST）设置峰段电流
V+	供应一个+5 V 电压
DOUT	串行数据输出引脚，此信号常用于几个 MAX7219 的级联

显示部分的操作按照 MAX7219 的操作时序要求进行即可，这里不作介绍。

7.4.4 硬件原理图的设计

硬件原理图设计如图 7-25 所示。

图 7-24（a）和（b）中分别是 DS18B20 传感器与单片机的连接部分和 MAX7219 显示部分，DS18B20 采用的是电源供电方式， V_{DD} 接电源 V_{CC} ，GND 接地，DQ 引脚作为信号线。MAX7219 与单片机的连接与 DS18B20 的连接相似，同样是 V_{DD} 接电源 V_{CC} ，GND 接地，DIN、LOAD、CLK 引脚作为信号线与单片机相连，实现数据传输。

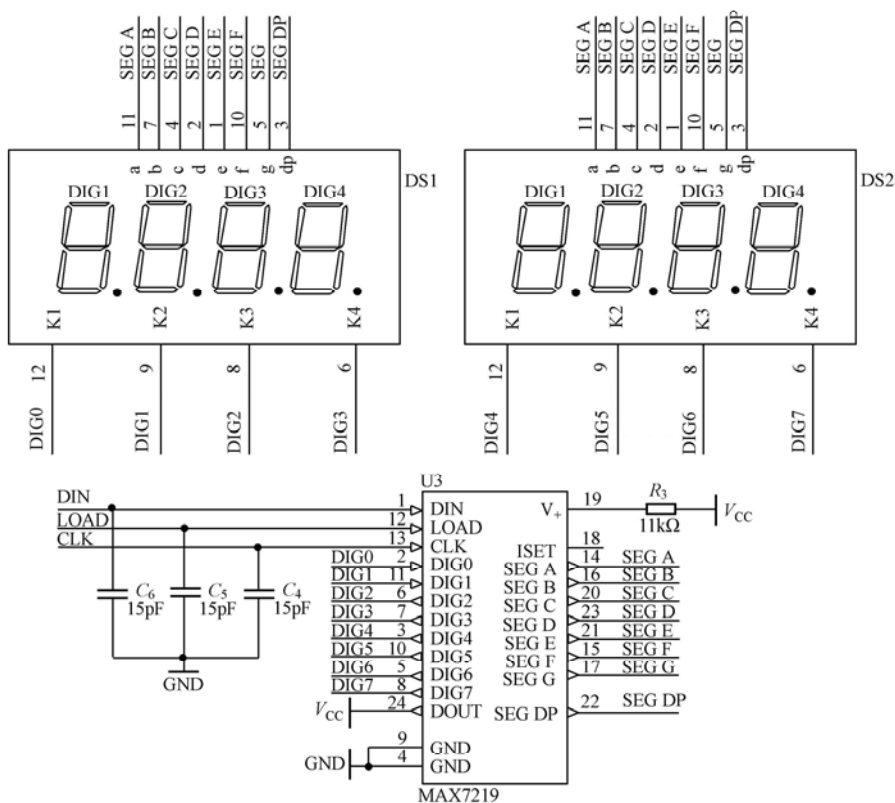
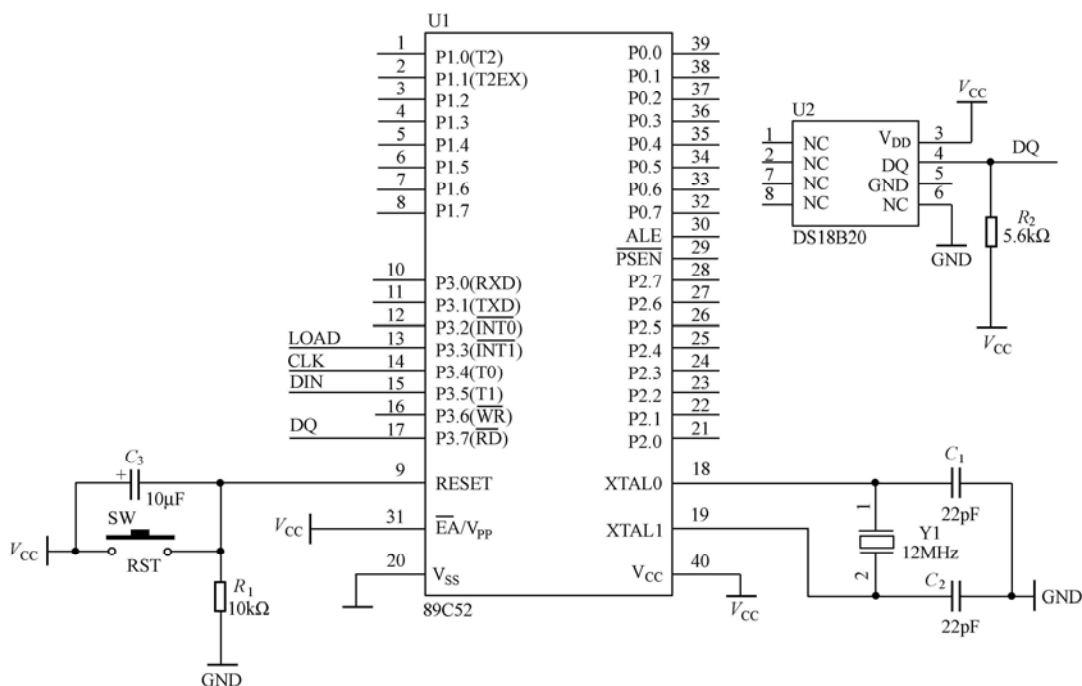


图 7-25 数字温度计电路

7.4.5 软件设计

通过对芯片功能的了解，绘制出程序流程图，并利用程序来显示其效果。

1. 程序流程

本例的程序流程图如图 7-26 所示。

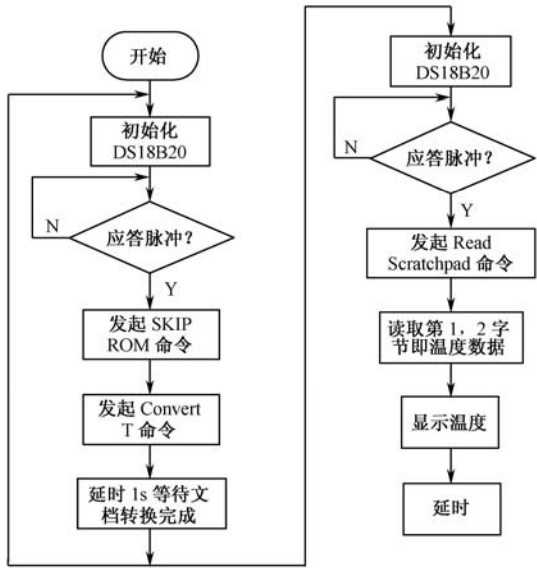


图 7-26 数字温度计程序流程图

在对正负温度的判断上，以 `hl` 为符号标志，初始值为 0，默认为正数。字符变量 `tpmsb` 的高五位判断正负温度，得到负数时将 `hl` 置 1。把变量 `tplsb` 的高四位和变量 `tpmsb` 的低三位通过位运算组合后存入字符变量 `temreg` 中，根据 `hl` 给温度的整数位和符号位赋值。由 `hl` 给小数位赋值，在处理小数部分时，将所得小数部分乘 625 后，对 10 求余。

2. 程序说明

因为 DS18B20 是单线器件，它在一根数据线上实现数据的双向传输，这就需要一定的协议来对读/写数据提出严格的时序要求，而 AT89C52 单片机并不支持单线传输。因此，必须采用软件的方法来模拟单线的协议时序。

本例程序说明如下：

```
#include<reg51.h> //引用标准库的头文件
#include<absacc.h>
#include<stdio.h>
#include<math.h>
typedef unsigned char uchar;
typedef unsigned int uint;
uchar tpls, tpmsb; //温度值低位，高位字节
uchar idata diss[8]={0,1,2,3,4,5,6,7};
```

```

uchar tempreg;
uchar coma ,comb;
uchar hl;
sbit din=P3^5;
sbit clk=P3^4;
sbit load=P3^3;
sbit DQ=P3^7;                                //数据通信线 DQ

```

//温度测量并显示程序

```

void disp(uchar d,uchar hh);
void w7219(uchar coma,uchar comb);
void sd7219(uchar trans);
void fenli(uchar shishu ,uchar xp );

```

//延时程序

```

void delay(uint t)
{
    uint i;
    while(t—)
    {
        //延时约 1ms
        for (i=0 ; i<125;i++)
        {}
    }
}

```

//产生复位脉冲初始化 ds18b20

```

void TxReset(void)
{
    uint i;
    DQ=0;
    /*拉低约 900 μs*/
    i=100;
    while(i>0)i—;
    DQ=1;
    i=4;
    while(i>0)i—;
}

```

//等待应答脉冲

```

void RxWait(void)
{
    uint i;
    while(DQ);
}

```

```

        while (~DQ);           //检测到应答脉冲
        i=4;
        while(i>0)i--;
    }
//读取数据的一个字节，满足写 1 和写 0 的时隙要求
bit RdBit (void)
{
    uint i;
    bit b;
    DQ=0;
    i++;
    DQ=1;
    i++;i++;
    b=DQ;
    i=8;
    while(i>0)i--;
    return(b);
}
//读取数据的一个字节
uchar RdByte(void)
{
    uchar i,j,b;
    b=0;
    for(i=1;i<=8;i++)
    {
        j=RdBit();
        b=(j<<7)|(b>>1);
    }
    return(b);
}
//写数据的一个字节，满足写 1 和写 0 的时隙要求
void WrByte(uchar b)
{
    uint i;
    uchar j;
    bit btmp;
    for(j=1;j<8;j++)
    {
        btmp=b&0x01;
        b=b>>1;           //取下一位（由低位到高位）
        if(btmp)

```

```

        {
            /*写 1*/
            DQ=0;
            i++;i++;          //延时，使得 15 μs 内拉高
            DQ=1;
            i=8;
            while(i>0)i--;    // 整个写 1 时隙不低于 60 μs
        }
    else
    {
        DQ=0;
        i=8;
        while(i>0)i--;
        DQ=1;
        i++;i++;
    }
}

//启动温度转换
void convert (void)
{
    TxReset();              //产生复位脉冲，初始化 DS18B20
    RxWait();               //等待 ds18b20 的应答信号
    delay(1);               //延时
    WrByte(0xcc);           //skip rom 命令
    WrByte(0x44);           //convert T 命令
}

//读取温度值
void RdTemp(void)
{
    TxReset();
    delay(1);
    WrByte(0xcc);
    WrByte(0xbe);
    tpls=RdByte();          //温度值的低位字节（低 4 位为小数部分）
    tpms=RdByte();          //温度值的高位字节（高 4 位为符号位）
}

void main(void)
{
    coma=0x0b;              //初始化 7219
    comb=0x07;

```

```

w7219(coma,comb);
coma=0x09;
comb=0xff;
w7219(coma,comb);
coma=0x0a;
comb=0x09;
w7219(coma,comb);
coma=0x0c;
comb=0x01;
w7219(coma,comb);
while(1)
{
    delay(1);                //延时 1 ms
    convert();
    delay(1000);

    hl=0;
    disp(tpmsb,1);           //发送高位字节，判断正负温度
    disp(tplsb,0);           //发送低位字节，显示温度
}
}

//显示温度
void disp(uchar d,uchar hh)
{
    uchar b=1;
    uchar *j;
    fenli (d,hh);
    j=diss;
    if(hh==0)                //转换完毕后，显示温度
    {
        for (b=1;b<9;b++,j++)
            {w7219(b,*j);    }
    }
}

void w7219(uchar coma,uchar comb)
{
    load=0;
    sd7219(coma);            //写地址

```



```

        sd7219(comb);                                //写数据
        load=1;
    }
    void sd7219(uchar trans)
    {
        uchar i;
        uchar j;
        for(i=0;i<8;i++)
        {
            clk=0;
            j=trans&0x80;
            trans<<01;
            if(j==0x00)din=0;
            else      din=1;
            clk=1;
        }
    }
    void fenli(uchar shishu,uchar xp)
    {
        uchar i,j,k;
        int m;
        i=shishu;
        if(xp==1)
        {
            if((i>>3)>0)
            {
                i=~i;i--;                                //负数时取补
                hl=1;                                    //标志为负数
                diss[0]=0xa;                            //显示负号
            }
            else
                diss[0]=0xf;                            //不显示
            tempreg=i<<4;
        }
        else
        {
            if (hl==1)
                {i=~i;i--;}                            //负数取补
            j=i&0xf;
            m=j*625;
            for(k=7;k>3;k--)                            //保存整数部分的 BCD 码

```

```

    {
        diss[k]=(uchar)(m%10);
        m=(int)(m/10);
    }
    j=i&0xf0;
    j>>=4;
    j=tempreg;
    for(k+=3;k>0;k--)          //保存小数部分的 BCD 码
    {
        diss[k]=j%10;
        j=(uchar)(j/10);
    }
    diss[3]=0x80;              //高 4 位后显示一个小数点
    if(diss[1]==0)    diss[1]=0xf;    //整数最高
    if(diss[1]==0xf&&diss[2]==0)    diss[2]=0xf;
}
}

```

7.5 基于双口RAM的单片机间通信

在高速数据采集系统以及在线测控系统中，为了实现数据采集、处理、实时控制和打印等功能，越来越多地采用双单片机系统。因此，采用专用双口 RAM 芯片是一种好的解决方案。

本例将介绍应用双口 RAM 实现双单片机并行通信系统的设计。

7.5.1 实例分析

双口 RAM 具有两组相互独立的地址线、数据线和控制线。本例将通过双口 RAM 实现两个单片机之间数据的传输，所选用的双口 ARM 芯片是 IDT 公司制造的 IDT7005。

7.5.2 IDT7005 芯片介绍

IDT7005 提供两套独立的端口，都有各自的地址总线、控制总线和 I/O 口。双口 RAM 允许两个处理器各访问一个端口，每个处理器都可将其看做自己的本地存储器，允许随机访问存储器的任何地址。L、R 分别表示左、右端口。D0~D7 为数据总线，A0~A12 为地址总线，SEM、CE、RD、WE、BUSY 分别表示使能、片选、读、写和忙标志。当 CE=SEM=1，禁止片选，芯片处于掉电备用状态。IDT7005 允许同时从同一地址读取数据，但是如果两端口同时向同一地址写操作或者进行读/写两种操作，则会发生冲突。为解决这一矛盾，在硬件上 IDT7005 自动产生 BUSY 信号表示冲突，该信号接到处理器 READY 端，使其读或写的时间延长。当 M/S=1，主模式 BUSY 为输出忙标志，而当 M/S=0，从模式 BUSY 为输入忙标志。

1. IDT7005 的引脚说明及内部框架

IDT7005 的引脚如图 7-27 所示。

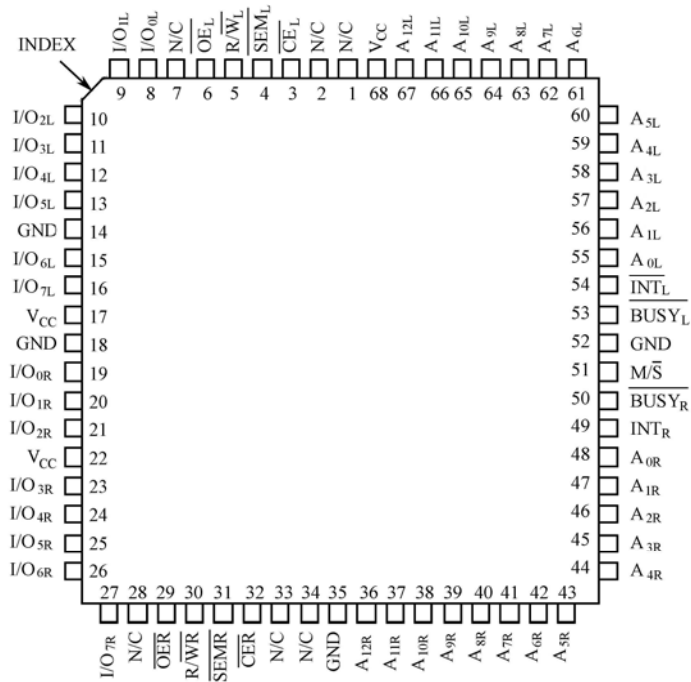


图 7-27 IDT7005 的引脚图

IDT7005 的引脚说明如表 7-9 所示。

表 7-9 IDT7005 的引脚说明

左 端 口	右 端 口	说 明
$\overline{\text{CE}}_{\text{L}}$	$\overline{\text{CE}}_{\text{R}}$	片选使能端
$\text{R}/\overline{\text{W}}_{\text{L}}$	$\overline{\text{W}}_{\text{R}}$	读/写使能端
$\overline{\text{OE}}_{\text{L}}$	$\overline{\text{OE}}_{\text{R}}$	输出使能端
$\text{A}_{0\text{L}}\sim\text{A}_{12\text{L}}$	$\text{A}_{0\text{R}}\sim\text{A}_{12\text{R}}$	地址线
$\text{I}/\text{O}_{0\text{L}}\sim\text{I}/\text{O}_{7\text{L}}$	$\text{I}/\text{O}_{0\text{R}}\sim\text{I}/\text{O}_{7\text{R}}$	数据输入/输出
$\overline{\text{SEM}}_{\text{L}}$	$\overline{\text{SEM}}_{\text{R}}$	使能端
$\overline{\text{BUSY}}_{\text{L}}$	$\overline{\text{BUSY}}_{\text{R}}$	忙标志位
$\overline{\text{INT}}_{\text{L}}$	$\overline{\text{INT}}_{\text{R}}$	中断标志位
$\text{M}/\overline{\text{S}}$		主/从选择
NC		悬空
V_{CC}		+5 V 电源
GND		接地

真值表如表 7-10 所示。

表 7-10 真值表

输 入				输 出	模 式
\overline{CW}	R/W	\overline{OE}	\overline{SEM}	I/O0~I/O7	
H	×	×	H	高阻态	掉电备用状态
L	H	×	H	DATA _{IN}	写模式
L	L	H	H	DATA _{OUT}	读模式
×	×	L	×	高阻态	输出无效

注：H 代表高电平；L 代表低电平；×代表任意电平；左端口地址线上的值不等于右端口地址线上的值。

IDT7005 的内部结构如图 7-28 所示。

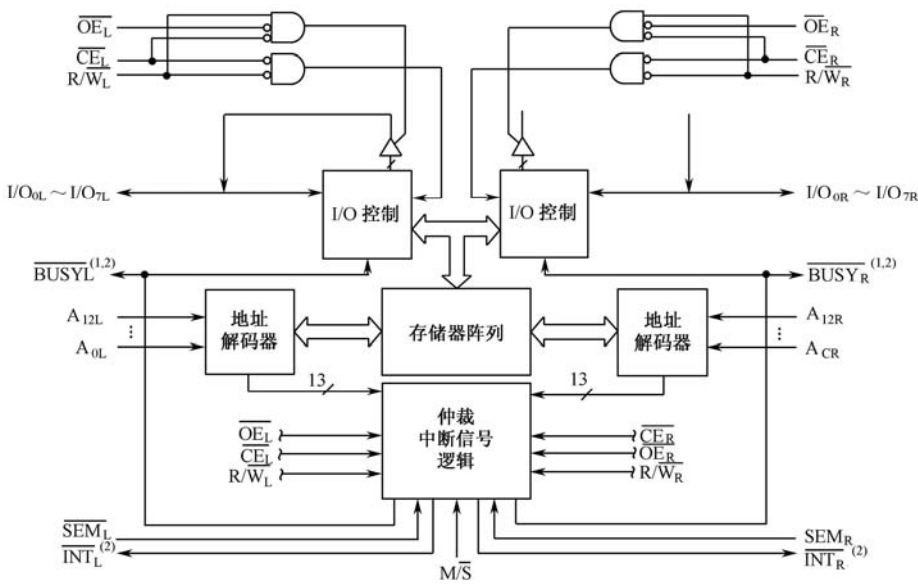


图 7-28 IDT7005 的内部结构

DeviceNet 适配器通过一个具有中断功能的双口 RAM IDT7005 提供与其他应用电路的通信接口。IDT7005 允许 2 个 CPU 对双口 RAM 的同一个单元在不同的时间进行读/写；具有 2 套完全独立的中断逻辑，实现 2 个 CPU 间的握手控制信号。IDT7005 的最高 2 个字节 1FFEh 和 1FFFh 分别兼做 2 个端口的中断逻辑单元。

2. IDT7005 的特性

IDT7005 的读周期时序如图 7-29 所示。

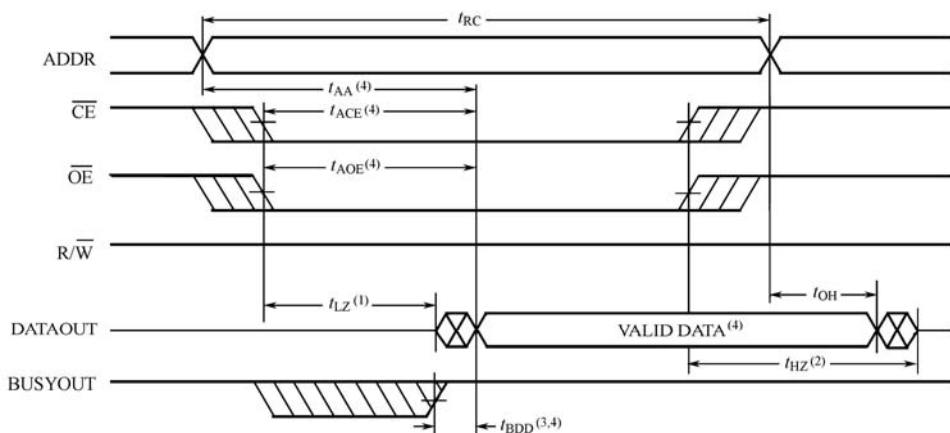


图 7-29 IDT7005 的读周期时序图

通过读周期时序图可以看出：

- $\overline{\text{OE}}$ 和 $\overline{\text{CE}}$ 中后有效的信号决定时序；
- $\overline{\text{OE}}$ 和 $\overline{\text{CE}}$ 中先无效的信号决定时序；
- 仅在反向端口对同一个地址进行写操作的时候才需要延时 t_{BDD} ，当两个端口同时读一个地址的数据时，忙标志位 $\overline{\text{BUSY}}$ 不会影响数据的有效传输；
- 只有当 t_{BDD} 、 t_{AOE} 、 t_{ACE} 、 t_{AA} 都有效时数据才会开始传输。

IDT7005 的写周期时序（ $\text{R}/\overline{\text{W}}$ 作为约束时序）如图 7-30 所示。

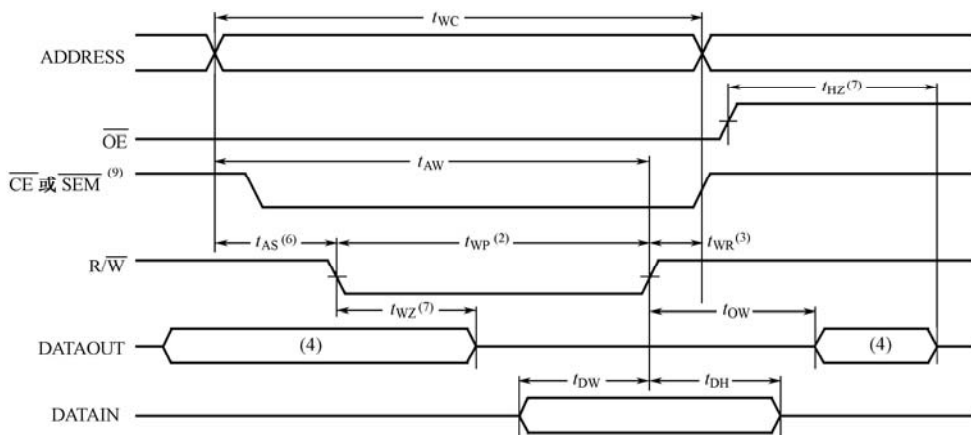


图 7-30 IDT7005 写周期的时序（ $\text{R}/\overline{\text{W}}$ 作为约束时序）

IDT7005 写周期的时序（ $\overline{\text{CE}}$ 作为约束时序）如图 7-30 所示。

通过图 7-30 和图 7-31 可以看出：

- 任何地址跳变时，作为约束时序的 $\text{R}/\overline{\text{W}}$ 或 $\overline{\text{CE}}$ 必须为高电平；
- 写操作只会发生在存储阵列写周期中 $\text{R}/\overline{\text{W}}$ 和 $\overline{\text{CE}}$ 都为低电平的交叠部分；
- $\text{R}/\overline{\text{W}}$ 和 $\overline{\text{CE}}$ 任意一个为高电平时 t_{WR} 开始，一直到写周期结束；
- 在写周期期间，I/O 口为输出状态并且输入信号绝对不可以使用；

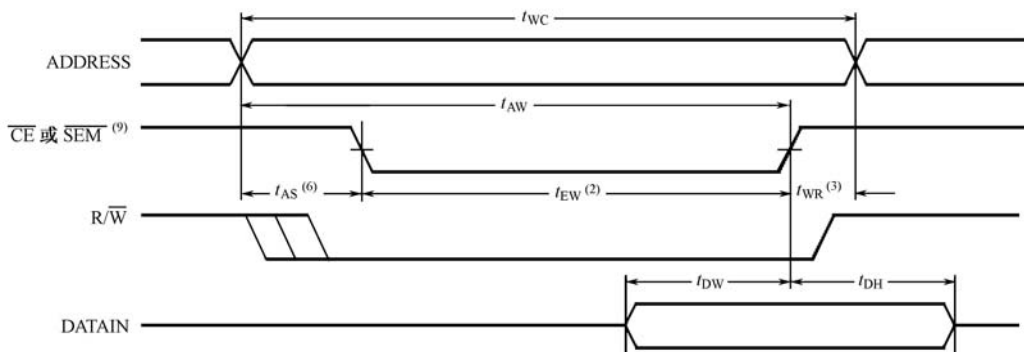


图 7-31 IDT7005 写周期的时序 (\overline{CE} 作为约束时序)

- 如果 \overline{CE} 的低跳变和 R/\overline{W} 的低跳变同时发生或在 R/\overline{W} 之后，输出就会保持在高阻态；
- R/\overline{W} 和 \overline{CE} 后有效的信号决定时序；
- 如果 R/\overline{W} 约束写周期时 \overline{OE} 为低电平， t_{DW} 要求写脉冲必须比 t_{WP} 或者 $(t_{WZ} + t_{DW})$ 宽，以便关闭 I/O 口驱动并且把数据传送到总线上；如果 R/\overline{W} 约束写周期时 \overline{OE} 为高电平，对 t_{DW} 的上述要求此时就没有必要，写脉冲也可以和规定的 t_{WP} 一样。

IDT7005 一些主要性能如下：

- 兼容 TTL 电平，+5 V 供电；
- 接入快，访问的最大时间在纳秒级；
- 功耗低，运行时为 750 mW，休眠时仅为 5 mW；
- 提供 BUSY 和中断标志；
- 硬件支持端口间的其余信号；
- 两端口间完全异步操作。

7.5.3 硬件设计

本例中的硬件电路由单片机和双口 RAM 两个大的部分组成。下面分别介绍两个部分的电路原理图。

1. 单片机部分的原理图

图 7-32 是对双口 RAM 左端操作的单片机电路原理图。对右端操作的单片机电路也是一样的。图中单片机采用的是 AT89C52，时钟为 11.0592 MHz。74LS373 是地址锁存器，它实现了系统的 13 位地址线的低 8 位地址线和 8 位数据线的时分复用，高 5 位的地址线由 P2.0~P2.4 口提供。P2.5 口接双口 RAM 左端口的旗语控制引脚，低电平时有效，可以对旗语空间操作；P2.6 口提供双口 RAM 的左端口片选信号，低电平时有效。

2. 双口RAM部分的原理图

双口 RAM 部分的原理图如图 7-33 所示，图中 R_{N1} 和 R_{N2} 为 1 k Ω 的电阻，它们和电阻 R_2 、 R_3 的作用是对双口 RAM 低电平有效的引脚 OEL 、 OER 、 R/WL 、 R/WR 、 $SEML$ 、 $SEMR$ 、 CEL 、 CER 、 $INTL$ 和 $INTR$ 实现上拉。IDT7005 的主/从选择引脚 M/S 接高电平，选择主模式。

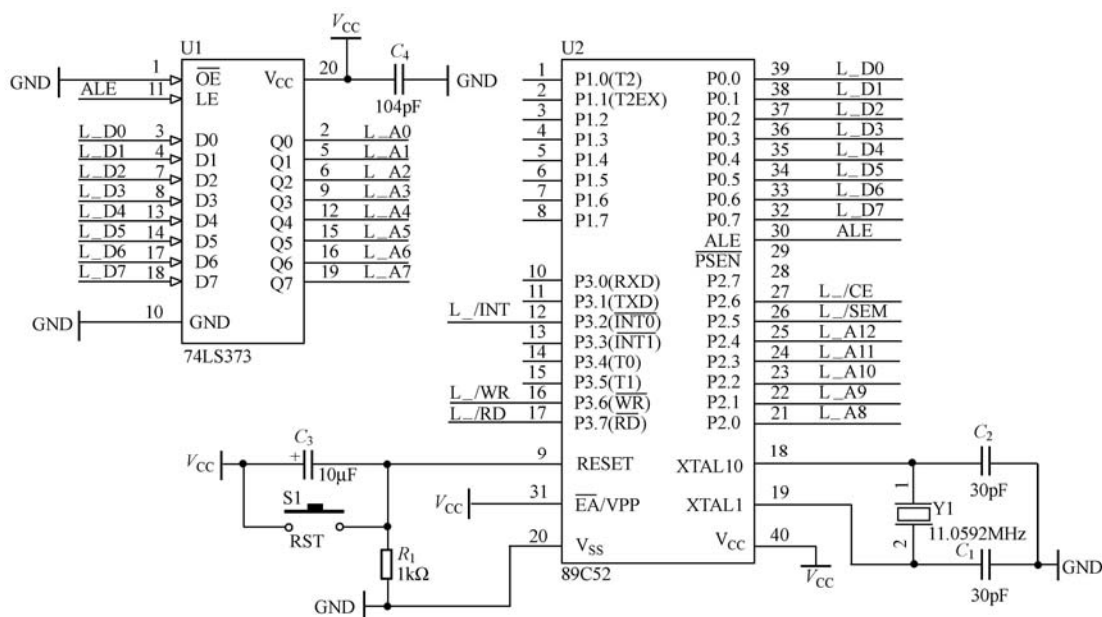


图 7-32 对双口 RAM 左端操作的单片机电路原理图

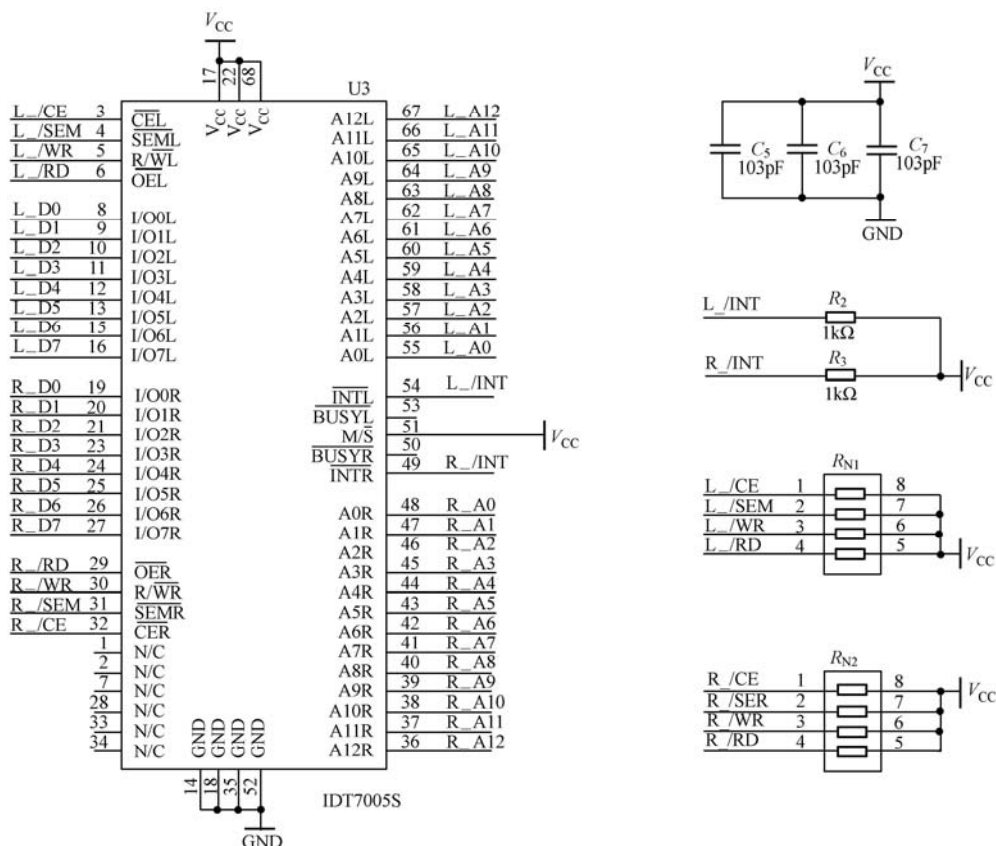


图 7-33 基于双口 RAM 的单片机间通信电路双口 RAM 部分原理图

7.5.4 软件设计

软件设计的时候要结合硬件的连线等设计作为参考。

1. 设计分析

IDT7005 有 13 位的地址线，所以存储器的容量为 2^{13}B ，即 8 KB。根据硬件设计电路所示的地址线、其旗语线、片选线的连接关系可知，IDT7005 的 RAM 地址空间为 2000H~3FFFH。

IDT7005 提供 8 个旗语标志，其地址线为 A0~A2，其他的高位地址都无效。对旗语操作时，P2.5 口和 P2.6 口需要同时为低电平（旗语控制位和旗语片选，均是低电平有效），所以 8 个其余标识的地址空间就是 0000H~0007H。本例中 IDT7005 的地址空间分为两个部分：2000H~2FFFH 为左端口单片机的存储空间，其中 2000H~24FFH 存放本机的状态信息，2500H~27FFH 存放本机对另一端单片机的配置命令信息，2800H~2FFFH 留做备用；3000H~3FFFH 为右端口单片机的存储空间，其分配原则和左端口的一样。

2. 程序代码

程序代码如下：

```
#include <reg52.h>                                // 引用标准库的头文件
#include <absacc.h>

#define uchar unsigned char

#define LP_STT_SEM XBYTE[0x0000]                    // 左端状态旗语
#define LP_PRO_SEM XBYTE[0x0001]                    // 左端配置旗语
#define RP_STT_SEM XBYTE[0x0002]                    // 右端状态旗语
#define RP_PRO_SEM XBYTE[0x0003]                    // 右端配置旗语
#define INTL_SEM XBYTE[0x0004]                      // 左中断旗语
#define INTR_SEM XBYTE[0x0005]                      // 右中断旗语

#define DPRAM_INTL XBYTE[0x2FFF]                    // 右端口中断
#define DPRAM_INTR XBYTE[0x2FFE]                    // 左端口中断

#define READY 11                                    // 就绪状态

uchar flag_int0;                                     // 外部中断 0 标志位
uchar flag_ready;                                    // 另一端就绪标志位
uchar Conf_Times;                                    // 配置次数

uchar xdata State_Array[254];
uchar xdata *LpStateRamAddr;                        // 双口 RAM 左端状态空间起始地址
```



```

uchar xdata *RpStateRamAddr;           // 双口 RAM 右端状态空间起始地址
uchar xdata *LpConfRamAddr;            // 双口 RAM 左端配置空间起始地址
uchar xdata *RpConfRamAddr;            // 双口 RAM 右端配置空间起始地址


void InitConfRP(void);
void Conf(void);
void State_Fill(void);
void State_Get(void);
bit get_sem(uchar *sem_type);


void main()
{
    flag_int0 = 0;
    flag_ready = 0;
    Conf_Times = 0;


    LpStateRamAddr = 0x2000;
    LpConfRamAddr = 0x2400;
    RpStateRamAddr = 0x3000;
    RpConfRamAddr = 0x3400;


    //等待右端就绪
    while(flag_ready!=1)
    {
        get_sem(&RP_STT_SEM);           // 申请右端状态旗语
        if (*RpStateRamAddr == READY)
            flag_ready = 1;              // 右端就绪标志置 1
        RP_STT_SEM = 0x01;              // 释放右端状态旗语
    }


    //右端初始配置
    InitConfRP();


    //向左端状态空间的第一地址单元写 READY，表示左端准备就绪
    get_sem(&LP_STT_SEM);                // 申请左端状态旗语
    *LpStateRamAddr = READY;             // 左端就绪
    LP_STT_SEM = 0x01;                  // 释放左端状态旗语


    Conf_Times++;

```

```

        EA = 1;                                // 开 CPU 中断
EX0 = 1;                                       // 开外部中断 0
        ET0 = 1;                               // 开 T/C0 中断
PX0 = 0;                                       // 外部中断低优先级
PT0 = 1;                                       // 计数器高优先级
TMOD = 0x01;                                  // 工作在方式 1
TH0 = 0x70;                                   // 预置定时器计数初值
TL0 = 0x00;
TR0 = 0;                                       // 禁止 T0

//右端接收左端的初始化配置，运行后触发双口 RAM 的左端中断，左端中断触发后对
//右端作第二次配置，并启动定时器，开始定期更新本机的状态信息和监测右端的状态
while(flag_int0==1)
{
    if (Conf_Times==1)
    {
        Conf_Times++;                          // 右端配置次数加 1
        Conf();                                // 对右端二次配置

        //触发右端中断，通知右端接收二次配置
        get_sem(&INTR_SEM);                    // 申请右中断旗语
        DPRAM_INTL = 0xFF;                     // 右端中断
        INTR_SEM = 0x01;                       // 释放右中断旗语
    }

    TR0 = 1;                                    // 定时器 T0
}
}

//申请旗语函数
bit get_sem(uchar *sem_type)
{
    *sem_type = 0x00;                          // 申请旗语
    while((*sem_type!=0x00));                  // 获得旗语
    return(1);
}

//对右端初始化配置函数
void InitConfRP(void)
{
    uchar i;

```

```

    get_sem(&LP_PRO_SEM);                // 申请左端配置旗语
    for (i=0;i++;i<=255)
        *(LpConfRamAddr+i) = 0x22;
    LP_PRO_SEM = 0x01;                    // 释放左端配置旗语
}

//对右端单片机的二次配置函数
void Conf(void)
{
    uchar i;
    get_sem(&LP_PRO_SEM);                // 申请左端配置旗语
    for (i=0;i++;i<=255)
        *(LpConfRamAddr+i) = 0x33;
    LP_PRO_SEM = 0x01;                    // 释放左端配置旗语
}

//更新本机状态函数
void State_Fill(void)
{
    uchar i;
    get_sem(&LP_STT_SEM);                // 申请并获得左端状态旗语
    for (i=0;i++;i<=254)
        *(LpStateRamAddr+i+1) = 0x44;
    LP_STT_SEM = 0x01;                    // 释放左端状态旗语
}

//查询另一端单片机状态函数
void State_Get(void)
{
    uchar i;
    get_sem(&RP_STT_SEM);                // 申请并获得右端状态旗语
    for (i=0;i++;i<=254)
        State_Array[i] = *(RpStateRamAddr+i+1);
    RP_STT_SEM = 0x01;                    // 释放右端状态旗语
}

//定时中断服务子程序,用于更新左端状态信息和查询右端状态信息
void timer0_int() interrupt 1 using 1
{
    TR0 = 0;                            // 关闭定时器 0

```

```

    TH0 = 0x70;                // 重置定时器计数
    TL0 = 0x00;
    State_Fill();               // 定期更新左端单片机状态让右端单片机可查询
    State_Get();                // 定期查询右端单片机的状态信息
}

//外部中断 0 服务子程序，设置中断标志位 flag_int0
void out_int0() interrupt 0 using 1
{
    uchar ch;
    flag_int0 = 1;              //双口 RAM 产生的中断
    get_sem(&INTL_SEM);         // 申请左中断旗语
    ch = DPRAM_INTR;            // 清中断
    INTL_SEM = 0x01;            // 释放左中断旗语
}

```

第8章 通信实例

随着单片机系统的应用和网络的发展，单片机通信功能显得越来越重要。单片机通信包括单片机与外部设备之间，也包括单片机和单片机之间的信息交换。本章将介绍几种典型的单片机通信实例。

8.1 单片机实现点对点的数据传输

所谓点对点传输就是串行通信，是在一根传输线上一位一位地传送信息。

8.1.1 实例说明

单片机在控制本地的外围期间时，通过 8 位并行数据总线进行信息的交互，单片机还需要控制远端的设备，可以利用单片机的串行通信模块 RS-232 接口延长传输距离。

本例的功能就是利用 RS-232 接口实现单片机之间的点对点传输。

8.1.2 串行通信

串行通信是指外设和单片机之间使用一根数据信号线（另外需要地线，可能还需要控制线），数据在一根数据信号线上一位一位地进行传输，每一位数据都占据一个固定的时间长度。与并行通信相比，这种通信方式使用的数据线少，在远距离通信中可以节约通信成本，传输距离长，缺点是其传输速度比并行传输速度慢。这种通信方式多用于 CPU 和外设之间的通信。

常用的串口有 9 针串口（DB9）和 25 针串口（DB25），通信距离较近时（小于 12m），可以用电缆线直接连接标准 RS-232 端口；距离较远时，需附加调制解调器（MODEM）。最常用的方法是三线制接法，即地、接收数据和发送数据三个引脚相连。

表 8-1 是 DB9 和 DB25 的常用信号引脚说明。

表 8-1 DB9 和 DB25 的常用信号引脚说明

9 针串口（DB9）			25 针串口（DB25）		
针号	功能说明	缩写	针号	功能说明	缩写
1	数据载波检测	DCD	8	数据载波检测	DCD
2	接收数据	RXD	3	接收数据	RXD
3	发送数据	TXD	2	发送数据	TXD
4	数据终端准备	DTR	20	数据终端准备	DTR
5	信号地	GND	7	信号地	GND
6	数据设备准备好	DSR	6	数据准备好	DSR
7	请求发送	RTS	4	请求发送	RTS
8	清除发送	CTS	5	清除发送	CTS
9	振铃指示	DELL	22	振铃指示	DELL

串行通信包括单工、半双工和全双工三种通信方式。单工通信是指信息只能从一端传到另一端，如收音机；半双工通信是指两端可以互相传输数据，但同一时刻只能存在一个方向的传输，如对讲机；全双工通信是指任意时刻两端都可以向对方传输信息，如手机。

串行通信可以分为两种类型：同步通信和异步通信。异步通信指字符一个一个地传输，每个字符一位一位地传输，传输一个字符时，从起始位开始，然后传输字符本身的各位，接着传输校验位，最后以停止位结束该字符的传输。一次传输的起始位、字符各位、校验位、停止位构成一组完整的信息，称为帧。帧与帧之间可有任意个空闲位。起始位之后是数据的最低位。同步通信指通信双方必须先建立同步，即双方的时钟要调整到同一个频率。收发双方不停地发送和接收连续的同步比特流。

1. RS-232C介绍

RS-232 标准是美国 EIA（电子工业联合会）与 Bell 等公司一起开发的、于 1969 年公布的通信协议。RS-232 被定义为一种在低速率串行通信中增加通信距离的单端标准。RS-232 属于单端通信，即采取不平衡传输方式。

RS-232C 总线标准设有 25 条信号线，包括一个主通道和一个辅助通道，当通信距离较近时，通常不需要 Modem，通信双方可以直接连接，这种情况下，只需使用少数几根信号线。最简单的情况，在通信中根本不需要 RS-232C 的控制联络信号，只需三根线（发送线、接收线、信号地线）便可实现全双工异步串行通信。RS-232C 标准规定，驱动器允许有 2 500 pF 的电容负载，通信距离将受此电容限制，例如，采用 150 pF/m 的通信电缆时，最大通信距离约为 15 m；若每米电缆的电容量减小，通信距离可以增加。传输距离短的另一原因是 RS-232C 属单端信号传送，存在共地噪声和不能抑制共模干扰等问题，因此一般用于 50 英尺（1 英尺=0.3048 m）以内的通信。

目前 RS-232 是 PC 与通信工业中应用最广泛的一种串行接口。RS-232C 标准规定的数据传输速率为每秒 50、75、100、150、300、600、1 200、2 400、4 800、9 600、19 200 波特。它的最大传输速率为 20 kb/s，最大驱动输出电压为 ± 25 V；驱动器负载阻抗为 3~7 k Ω ；摆率的最大值为 30 V/ μ s；接收器输入电压范围是-15~+15 V；接收器输入门限是-3~+3 V；接收器输入电阻为 3~7 k Ω ；驱动器共模电压是-3~+3 V；接收器共模电压为-7~+7 V。

2. 接口电平转换

当两个 C51 单片机近距离通信时，可以将它们自带的串口相连，这样就可以完成数据的传输。具体做法就是：将一方的 TXD 和另一方的 TXD 引脚相连，将它们的接地引脚连在一起并接地。

当通信距离超过一定长度，便可以用 RS-232 接口延长通信距离。值得注意的是 RS-232 标准规定的逻辑电平与单片机（TTL 等数字电路）的逻辑电平不兼容，这时候就要引出一款可以实现电平转换的集成芯片。

本例中使用的是 MAXIM 公司生产的 MAX3232，它可以实现 EIA-232 接口的低功耗电平转换，包含两路收发器，具有静电保护功能而且数据传输速率可以保证在 25 kb/s 以上。

8.1.3 MAX3232 芯片介绍

本例采用 MAXIM 公司生产的 MAX3232 芯片，用于实现 TTL 电平和 RS-232 电平的转换。MAX3232 供电电压 5 V 或 3.3 V，耗电 0.3 mA，外接 4 个 0.1 μ F 电容，是 MAX232 的改进型。它的优势在于低功耗，传输速率可以达到 1 Mb/s，其引脚如图 8-1 所示。

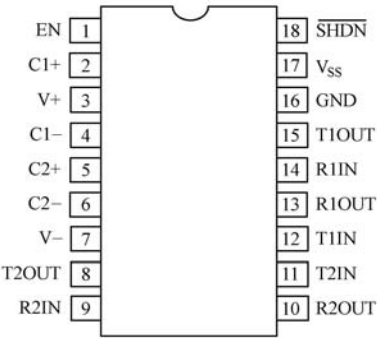


图 8-1 MAX3232 引脚图

各引脚功能说明如表 8-2 所示。

表 8-2 MAX3232 引脚功能说明

引 脚 名 称	说 明
V _{CC}	电源端，供电范围：3.3~5.5 V
GND	接地端
C1 ⁺ , C1 ⁻	电压加倍充电泵电容的正，负端
C2 ⁺ , C2 ⁻	转化充电泵电容的正，负端
V ⁺ , V ⁻	充电泵产生的±5.5 V 电压
T2IN, T1IN	TTL/CMOS 发送器输入
T2OUT, T1OUT	RS-232 发送器输出
R2IN, R1IN	RS-232 接收器输入
R2OUT, R1OUT	TTL/CMOS 接收器输出

1. MAX3232 的外围电路

MAX3232 的外围电路如图 8-2 所示。

图中的 C₁、C₂、C₃、C₄ 的容值都是 0.1 μ F。需要注意的是 MAX3232 有两路收发器，使用时应视具体情况而定（注：C₃ 的另一端可以接到任意的地或 V_{CC} 端）。

2. MAX3232 的特征曲线

MAX3232 的特征曲线包括电容装载容量-发射器输出电压曲线、电容装载容量-回转比率曲线及数据传输时的电容装载容量-补给电流曲线等。

① 电容装载容量-发射器输出电压曲线如图 8-3 所示。

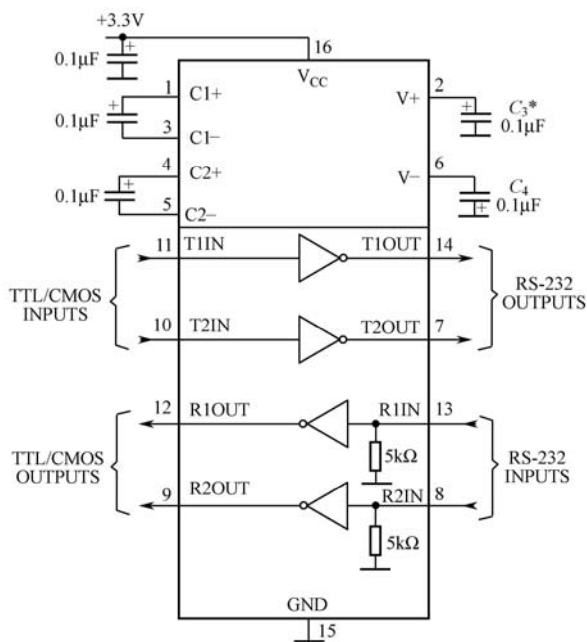


图 8-2 MAX3232 的外围电路图

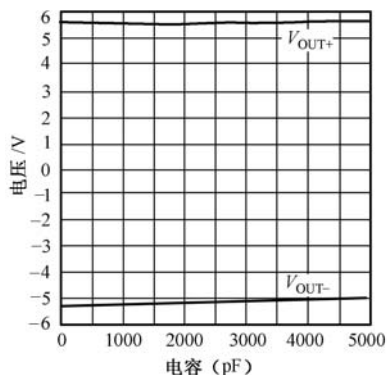


图 8-3 电容装载容量-发射器输出电压曲线

- ② 电容装载容量-回转比率曲线如图 8-4 所示。
 ③ 数据传输时的电容装载容量-补给电流曲线如图 8-5 所示。

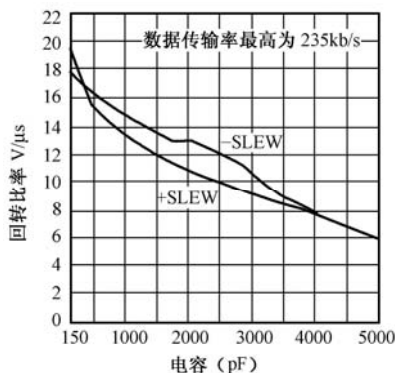


图 8-4 电容装载容量-回转比率曲线

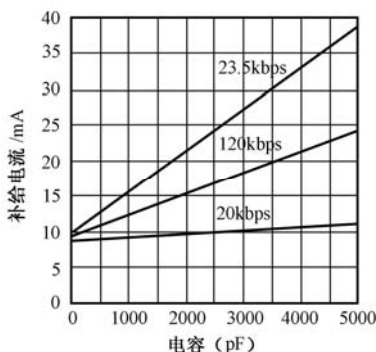


图 8-5 数据传输时的电容装载容量-补给电流曲线

8.1.4 硬件原理图的设计

图 8-6 是 MAX3232 实现的接口电平转换电路的原理图。

本例所实现的是两个单片机通过 RS-232 协议进行串口通信。这里仅使用了 MAX3232 两路收发器中的一路，其 R1OUT 和 T1IN 分别和单片机的 RXD (P3.1 口)、TXD (P3.0 口) 相连，而 T1OUT 和 R1IN 则通过 9 针串口 (DB9) 连接器 and 数据传输的另一方连接。值得注意的是，双方 DB9 连接器的 2、3 引脚要交叉连接，即一方的 T1OUT 和另一方的 R1IN 相连。

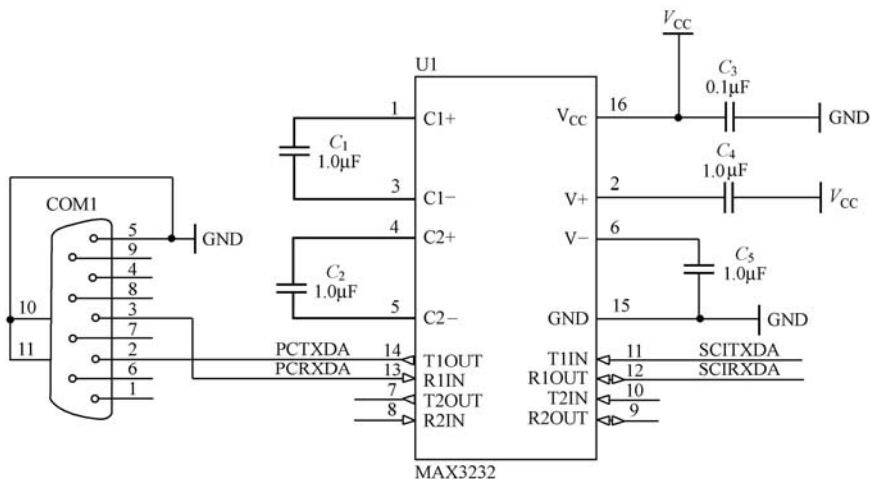


图 8-6 单片机点对点数据传输系统接口电平转换电路原理图

图 8-7 是单片机部分的电路原理图。

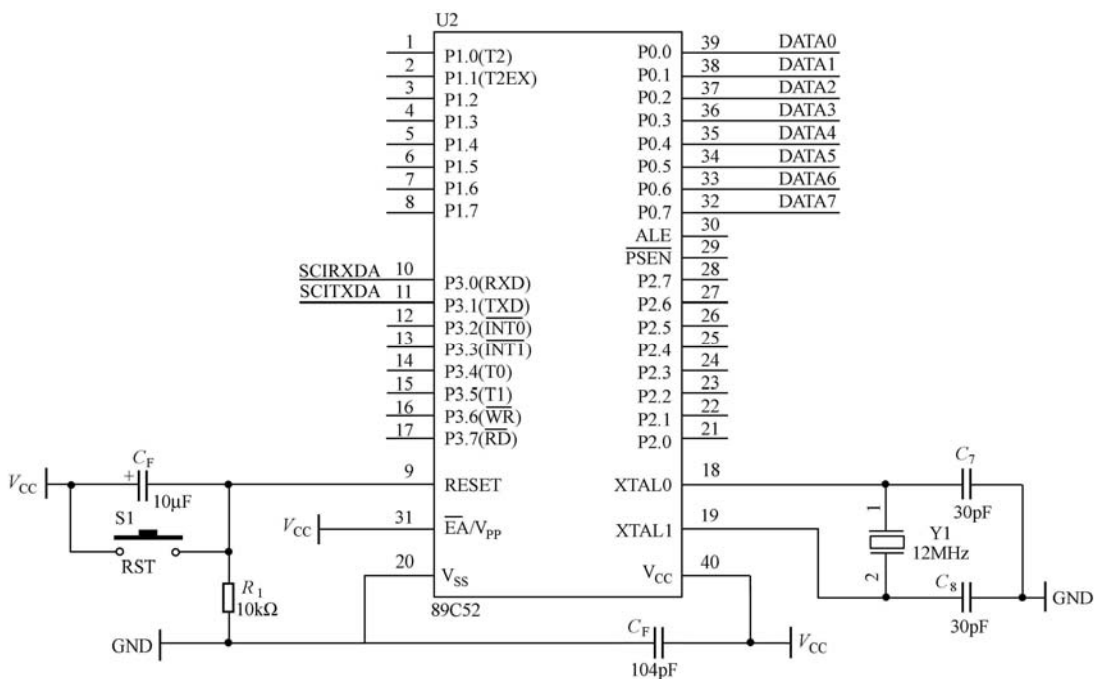


图 8-7 单片机点对点数据传输系统单片机部分电路原理图

图中 U2 为单片机芯片 AT89C52，它的时钟频率为 12 MHz，因此它的串口传输波特率的设置也已经被确定。单片机的 TXD (P3.0 口) 和 TXD (P3.1 口) 和电平转换芯片 MAX3232 连接，它们是单片机的串行通信的输入/输出信号。这样，本电路的连接就可以满足点对点传输要的要求，但就完整性而言，本例中还增加了 P0 口的使用。

发端：P0 口用于主机的数据采集。通过读取平底内容初始化发送数据缓冲区，每个

100 ms 读取一次，当读到 00H 就表示数据已经读取完毕。

收端：P0 口用来判断从机是否出于忙碌状态。当 P0 读到 BBH 时，就表示当前从机忙，需要向主机发送忙应答 15H。

8.1.5 软件设计

通过对芯片功能的了解，绘制出程序流程图，并利用程序来显示其效果。

1. 主机程序流程及代码

主单片机上的程序流程如图 8-8 所示。

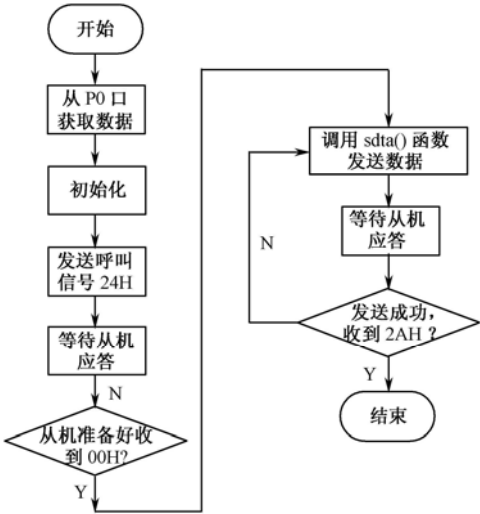


图 8-8 主机程序流程图

```
#include <reg51.h>
#include <string.h>

#define CALL 0x24           // 主机呼叫
#define BUSY 0x15          // 从机忙
#define OK 0x00             // 从机准备好
#define SUCC 0x2A          // 接收成功
#define ERR 0xF0           // 接收错误
#define MAXLEN 64          // 缓冲区最大长度
unsigned char buf[MAXLEN];

//延时 m ms
void delay(unsigned int m)
{
    unsigned int n;
```

```

while(m--)
{
    for (n=0;n<125;n++)
    {}
}

//发送数据函数
void sdta(unsigned char *buf)
{
    unsigned char i;
    unsigned char length;           //数据长度
    unsigned char ecc;             //校验字节

    length = strlen(buf);           //计算发送数据长度
    ecc = length;                   //校验字节计算

    //发送数据长度
    TI = 0;
    SBUF = length;                  //发送长度
    while(!TI);
    TI = 0;

    //发送数据
    for (i=0;i< length;i++)
    {
        ecc = ecc^(*buf);           //异或运算
        SBUF = *buf;
        buf++;
        while(!TI);
        TI = 0;
    }

    //发送校验字节
    SBUF = ecc;
    while(!TI);
    TI = 0;
}

//串口初始化函数
void init_serial()

```

```

{
    TMOD = 0x20;                // 定时器 T1 使用工作方式 2
    TH1 = 250;
    TL1 = 250;
    TR1 = 1;                    // 计时开始
    PCON = 0x80;                // SMOD = 1
    SCON = 0x50;                // 工作方式 1, 允许接收, 9 600 kb/s
}

void main()
{
    unsigned char i = 0;
    unsigned char tmp;
    tmp = BUSY;
    // 为缓冲区赋初值
    P0 = 0xff;
    while(P1 != 0)                // 从 P0 口读取, 0 表示数据采集结束
    {
        *(buf+i) = P0;
        delay(100);              // 延时 100 ms
        P0 = 0xff;
        i++;
    }
    *(buf+i) = 0;                // 缓冲区为 0 表示数据结束

    init_serial();               // 串口初始化

    EA = 0;                     // 关闭所有中断

    // 发送呼叫信号和接收应答信息, 如果没有接收到从机准备好的信号, 则重新发送呼叫帧
    tmp = BUSY;
    while(tmp != OK)
    {

        TI = 0;
        SBUF = CALL;
        while(!TI);
        TI = 0;                  // 发送呼叫

        RI = 0;
        while(!RI);
    }
}

```

```

        tmp = SBUF;
        RI = 0;                                //接收应答
    }

//发送数据和接收校验信息，如果接收到 SUCC，表示从机接收成功，否则将重新发送该组数据
    tmp = ERR;
    while(tmp!=SUCC)
    {
        Sdta(buf);        //发送数据
        RI = 0;
        while(!RI);
        tmp = SBUF;
        RI = 0;
    }
}

```

2. 从机程序流程及代码

从单片机的程序流程如图 8-9 所示。

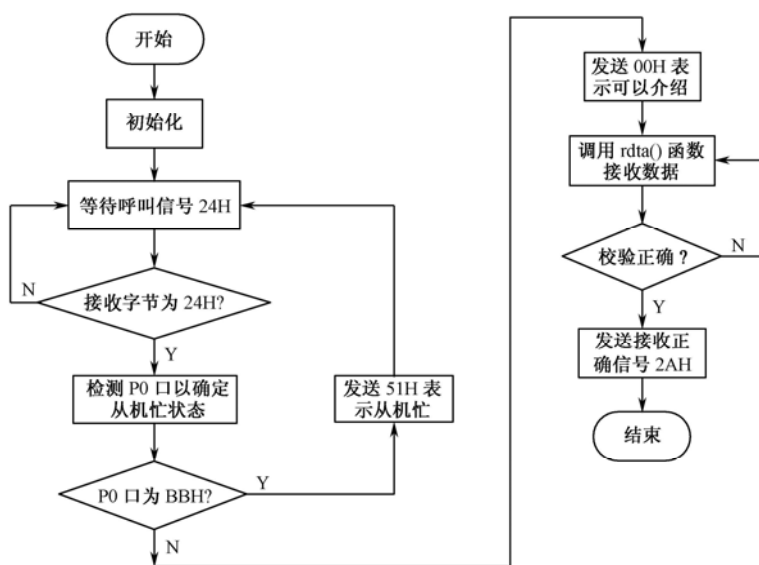


图 8-9 从机的程序流程图

从机程序代码如下：

```

#include <reg51.h>
#include <string.h>

```

```

//发送数据函数
unsigned char rdta(unsigned char *buf)
{
    unsigned char i;
unsigned char tmp;
    unsigned char length;           // 保存数据长度
    unsigned char ecc;              // 保存校验字节

    //接收数据长度
    RI = 0;
    while(!RI);
    length = SBUF;
    RI = 0;

    // length 的值为校验字节 ecc 初值
    ecc = length;

    //接收数据
    for (i=0;i< length;i++)
    {
        while(!RI);
        *buf = SBUF;
        ecc = ecc^(*buf);           //字节校验
        RI = 0;
        buf++;
    }
    *buf = 0;                       //表示数据结束

    //接收校验字节
    while(!RI);
    tmp = SBUF;
    RI = 0;

    //进行数据校验
    ecc = tmp^ecc;
    if (ecc!=0)
    {
        *(buf-length) = 0;         //清空数据缓冲区

        //发送校验错误信号 ERR
        TI = 0;
    }
}

```

```

        SBUF = ERR;
        while(!TI);
        TI = 0;

        return 0xff;                //表示校验错误
    }

    //发送校验成功信号 SUCC
    TI = 0;
    SBUF = SUCC;
    while(!TI);
    TI = 0;

    return 0;                        // 校验成功，返回 0
}

void init_serial()
{
    TMOD = 0x20;                    // 定时器 T1 使用工作方式 2
    TH1 = 250;
    TL1 = 250;
    TR1 = 1;                        // 开始计时
    PCON = 0x80;                    // SMOD = 1
    SCON = 0x50;                    // 工作方式 1，允许接收, 9 600 kb/s
}

void main()
{
    unsigned char tmp = 0;
    //串口初始化
    init_serial();

    EA = 0;                         //关闭所有中断

    while(1)
    {
        //如果接收到的数据不是 CALL，继续等待
        while (tmp!=CALL)
        {
            RI = 0;
            while(!RI)

```

```

        tmp = SBUF;
        RI = 0;
    }

    //检测 P0 口，如果 P0=0xBB，则为忙状态
    P0 = 0xff;
    tmp = P0;
    if(tmp==0xBB)                //如果 P0 口为 0xBB，发送 BUSY 信号
    {
        TI = 0;
        SBUF = BUSY;
        while(!TI);
        TI = 0;
        continue;
    }

    //发送 OK 信号，从机可以接收数据
    TI = 0;
    SBUF = OK;
    while(!TI);
    TI = 0;

    //数据接收
    tmp = 0xff;
    while(tmp==0xff)
    {
        tmp = rdata(buf);        //校验失败返回 0xff，接收成功返回 0
    }
}
}

```

8.2 单片机实现短距离无线通信

随着通信和信息技术的不断发展，短距离无线通信技术的应用步伐不断加快，正日益走向成熟。短距离无线通信泛指在较小的区域内（数百米）提供无线通信的技术，目前常见的技术有 802.11 系列无线局域网、蓝牙、HomeRF 和红外传输等技术。

8.2.1 nRF401 介绍

nRF401 是挪威 Nordic 公司新推出的一款集发射和接收为一体的无线数据传输芯片。nRF401 的 ISM 频段在 433 MHz，是单片机 UHF 无线收发芯片，满足欧洲电信工业标准（ETSI）EN300

200-1 V1.2.1。它采用 FSK 调制解调技术，最高工作速率可达 20 kb/s，发射功率可以调整，最大为 10 dBm。nRF401 集成度高，工作频率稳定可靠、外围元器件少、功耗极低，适合便携式产品的应用。nRF401 的天线接口设计为差分天线，以便于使用低成本的 PCB 天线。

其主要性能如下：

- 工作频率为国际通用的数传频段；
- 采用 FSK 调制，抗干扰能力强，特别适合工业控制场合；
- 采用 PLL 频率合成技术，频率稳定性极好；
- 灵敏度高，达到-105 dBm；
- 功耗小，接收状态 250 μ A，待机状态仅为 8 μ A；
- 低工作电压（2.7 V），可满足低功耗设备的要求；
- 具有多个频道，可方便地切换工作频率；
- 工作速率最高可达 20 kb/s；
- 仅外接一个晶体振荡器和几个电阻、电容、电感元件，基本无需调试。

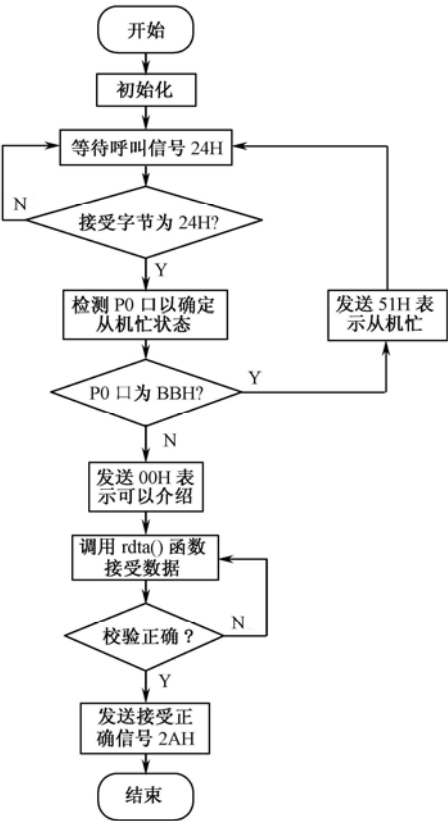


图 8-10 nRF401 的内部结构框图

1. nRF401 的内部结构及引脚说明

内部结构包括发射电路、接收电路、模式和低功耗控制逻辑电路及串行等接口几个部分，其中发射电路又包含射频功率放大器、锁相环（PLL），压控振荡器（VCO），频率合成器等电路。基准振荡器采用外接晶体振荡器，产生电路所需的基准频率。

nRF401 的内部结构（包括外围设备）如图 8-10 所示。

nRF401 的引脚如图 8-11 所示。

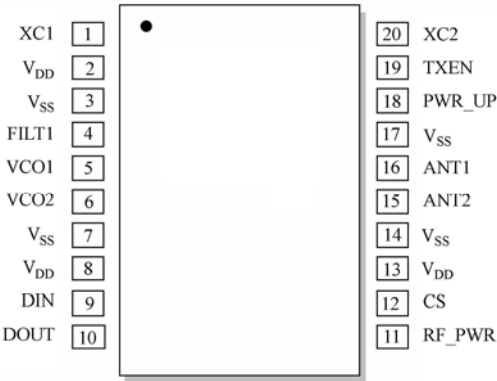


图 8-11 nRF401 的引脚图

各引脚功能说明如表 8-3 所示。

表 8-3 nRF401 引脚说明

引脚号	引脚名	引脚功能	说明
1	XC1	输入	晶振输入
2	V _{DD}	电源	提供电源（3~5 V 直流）
3	V _{SS}	电源地	接地
4	FILT1	输入	外接换装滤波器
5	VCO1	输入	外接压控振荡电感
6	VCO2	输入	外接压控振荡电感
7	V _{SS}	电源地	接地
8	V _{DD}	电源	提供电源（3~5 V 直流）
9	DIN	输入	数据输入
10	DOUT	输出	数据输出
11	RF_PWR	输入	发射功率设置
12	CS	输入	频段选择
13	V _{DD}	电源	提供电源（3~5 V 直流）
14	V _{SS}	电源地	接地
15	ANT2	输入/输出	外接天线
16	ANT1	输入/输出	外接天线
17	V _{SS}	电源地	接地
18	PWR_UP	输入	待机控制位
19	TXEN	输入	工作模式切换
20	XC2	输出	晶振输出

2. nRF401 工作状态切换

nRF401 包括 RX（接收模式）、TX（发送模式）和 Standby（待机模式）三种工作模式。

① 当从 RX 切换到 TX，即接收模式转换成发送模式时，数据输入脚 DIN 必须保持至少 1 ms 才能传输数据，其时序如图 8-12 所示。

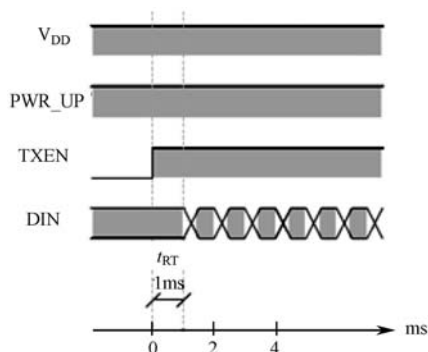


图 8-12 RX 切换至 TX 时序图

从 RX 切换到 TX 时，所需要的转换时间定义成 t_{RT} ，是从 TXEN 置 1 到 DIN 有效输出

的这段时间。 t_{RT} 至少是 1 ms。

② 当从 TX 切换到 RX，即发送模式转换成接收模式时，数据输出脚 DOUT 要至少保持 3 ms 才能有数据输出，其时序如图 8-13 所示。

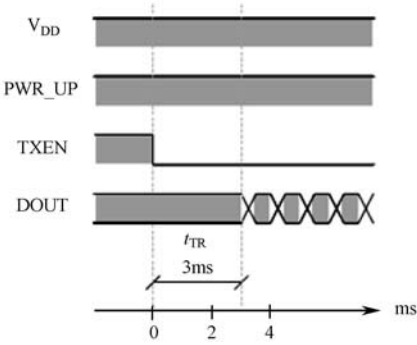


图 8-13 TX 切换至 RX 时序图

从 TX 切换到 RX 时，所需要的转换时间定义成 t_{TR} ，是从 TXEN 置 0 到 DOUT 有效输出的这段时间， t_{TR} 至少是 3 ms。

③ 当从 Standby 模式到 RX 模式时，当 PWR_UP 输入设成 1 时，经过 3 ms 时间后，DOUT 脚输出数据才有效。其时序如图 8-14 所示。

从 Standby 切换到 RX 时，所需要的转换时间定义成 t_{SR} ，是从 PWE_UP 置 1 到 DOUT 有效输出的这段时间。 t_{SR} 至少是 3 ms。

④ 从 Standby 模式到 TX 模式，所需稳定的最大时间不能确定，因为它需要两个状态控制位 (PWR_UP 和 TXEN) 同时作用。对 nRF401 来说， t_{ST} 最长的时间是 3 ms。其时序图如图 8-15 所示。

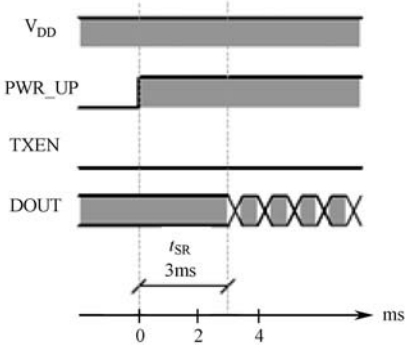


图 8-14 Standby 切换至 RX 时序图

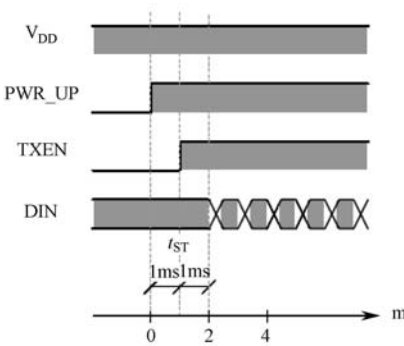


图 8-15 Standby 切换至 TX 时序图

从 Standby 切换到 TX 时，所需要的转换时间定义成 t_{ST} ，是从 PWE_UP 置 1 到合成频率稳定的这段时间， t_{ST} 至少是 3 ms。

⑤ 从上电到 TX 模式，为了避免开机时产生干扰和辐射，在上电过程中，TXEN 的输入引脚必须保持为低电平，这样才能使频率合成器进入稳定工作状态。TXEN 必须保持 1 ms 才可以往 DIN 发送数据。

⑥ 从上电到 RX 模式，不接收数据，DOUT 也不会有数据输出，直到电压稳定达到

2.7 V 以上，并且至少保持 5 ms，如果采用外部振荡器，时间可以缩短到 3 ms。

8.2.2 PTR2000 的介绍

PTR2000 是基于 nRF401 芯片的超小型、超低功耗、高速率（19.2 kb/s）无线数据传输模块，集发射和接收于一体。

1. PTR2000 模块性能

- 接收和发射集为一体；
- 工作频率为国际通用的数传频段 433 MHz；
- 采用 FSK 调制，抗干扰能力强，特别适合工业控制场合；
- 采用 DDS+PLL 频率合成技术，频率稳定性极好；
- 灵敏度高，可达 -105 dBm；
- 最大发射功率可达 +10 dBm；
- 低工作电压（2.7 V），功耗小，待机状态仅为 8 μ A；
- 具有两个频道，可以满足需要多信道工作的特殊场合；
- 工作速率最高可达 20 kb/s；
- 超小体积，约 40 mm \times 27 mm \times 5 mm；
- 可直接连接 CPU 串口使用，软件编程非常方便；
- 由于采用了低发射功率、高接收灵敏度的设计，使用无需申请许可证；
- 标准 DIP 引脚间距，更适合用于嵌入式设备。

2. 电气特性

PT2000R 的电气特性如表 8-4 所示。

表 8-4 PTR2000 的电气特性

参 数	数 值
工作频率	433.92 MHz/434.33 MHz
调制方式	FSK
温频方式	DDS+PLL
最大发射功率@3 V 400 Ω	+10 dBm
接收灵敏度@400 Ω 20 kb/s	-105 dBm
最高通信速率	20 kb/s
工作电压	2.7~5.25 V
电流	发射：20~30 mA；接收：10 mA
待机电流（PWR=0）	8 μ A

3. 引脚说明

PRT2000 的引脚如图 8-16 所示。

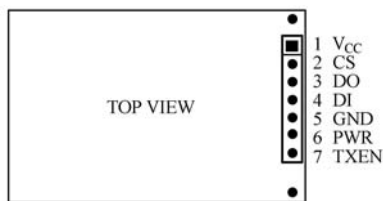


图 8-16 PRT2000 的引脚图

PRT2000 的引脚功能说明如表 8-5 所示。

表 8-5 引脚功能说明

引 脚 名 称	引 脚 说 明
V _{CC}	正电源，接 2.7~5.25 V
CS	频道选择
DO	数据输出
DI	数据输入
GND	电源地
PWR	节能控，PWR=1 时是正常工作状态，PWR=0 时是待机低功耗状态
TXEN	发射/接收控制

4. PTR2000 工作模式的设置

PTR2000 工作模式的设置如表 8-6 所示。

表 8-6 PTR2000 工作模式的设置

引 脚 电 平	工 作 模 式	
	工作频道/MHz	工作状态
TXEN/CS/PWR 001	1 (433.92)	接收 0
011	2 (434.33)	接收
101	1 (433.92)	发射
111	2 (434.33)	发射
**0	*	待机

注：*代表任意（0 或 1）

8.2.3 硬件设计

本例的硬件设计分别包括单片机和 PTR2000 接口电路设计和 PC 和 PTR2000 接口电路设计。

1. 单片机和PTR2000 接口电路设计

本例中单片机的电平通过串口以及 I/O 口与 PTE2000 直接相连，接口电路如图 8-17 所示。

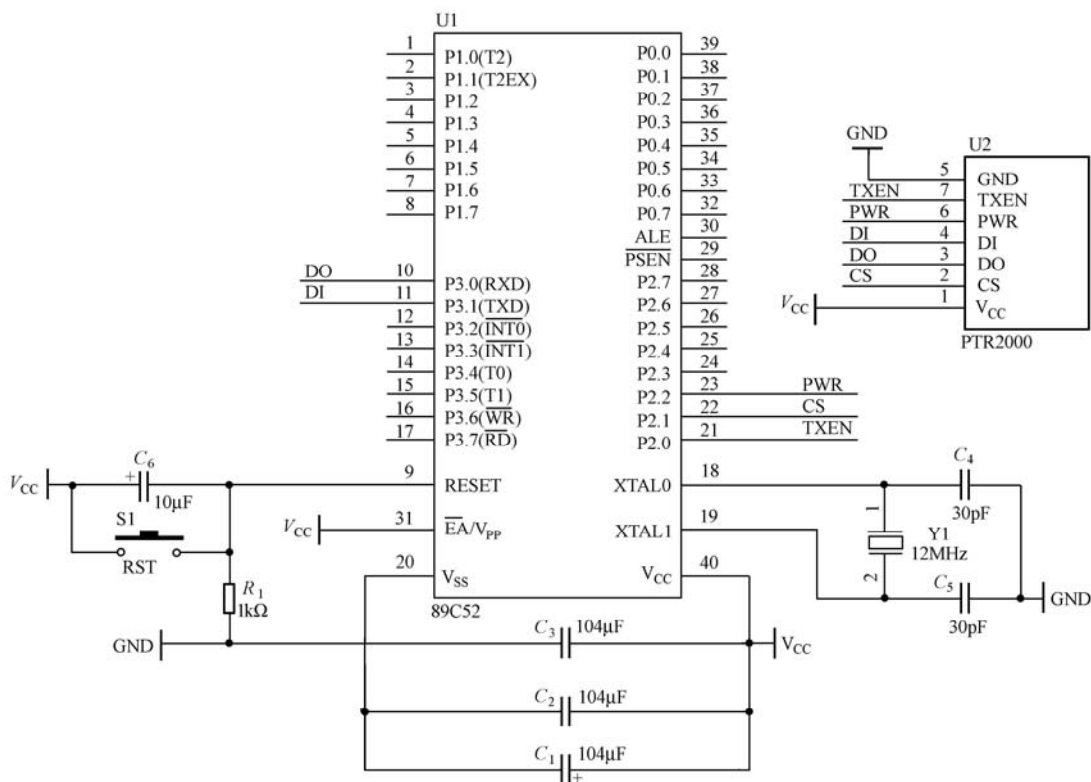


图 8-17 单片机和 PTR2000 接口电路原理图

S1 是复位键，与 C_3 、 R_1 构成了复位电路；Y1 是 12 MHz 的晶振，决定了单片机串口的传输速率；单片机的 P3.0 口和 PTR2000 的 DO 引脚相连，单片机的 P3.1 口和 PTR2000 的 DI 引脚相连，实现串行数据传输；PTR2000 的 TXEN、CS、PWR 三个引脚分别和单片机 I/O 口中的 P2.0 口、P2.1 口、P2.2 口相连。

2. PC和PTR2000 接口电路设计

PC 和 PTR2000 接口电路的原理如图 8-18 所示。

由于 PC 的串口支持 RS-232 标准，而 PTR2000 模块支持的是 TTL 电平，所以需要 MAX202 来完成 PC 和 PTR2000 之间的电平转换。

PC 的串口通过 MAX202 与 PTR2000 模块的串行输入，输出引脚 DI、DO 相连；PTR2000 的低功耗控制引脚 PWR 接 V_{CC} 高电平，让其处在工作状态；频道选择引脚 CS 接 GND 高电平，即采用通信频道 1 (433.92 MHz)；TXEN 引脚由 PC 串口的 RTS 信号来控制，决定了 PTR2000 模块的发送/接收状态， C_7 、 C_8 、 C_9 、 C_{10} 的电容值都是 $0.1\mu\text{F}$ 。

PC 端的串口传输速率也需设定为 96 kb/s，和单片机保持一致。

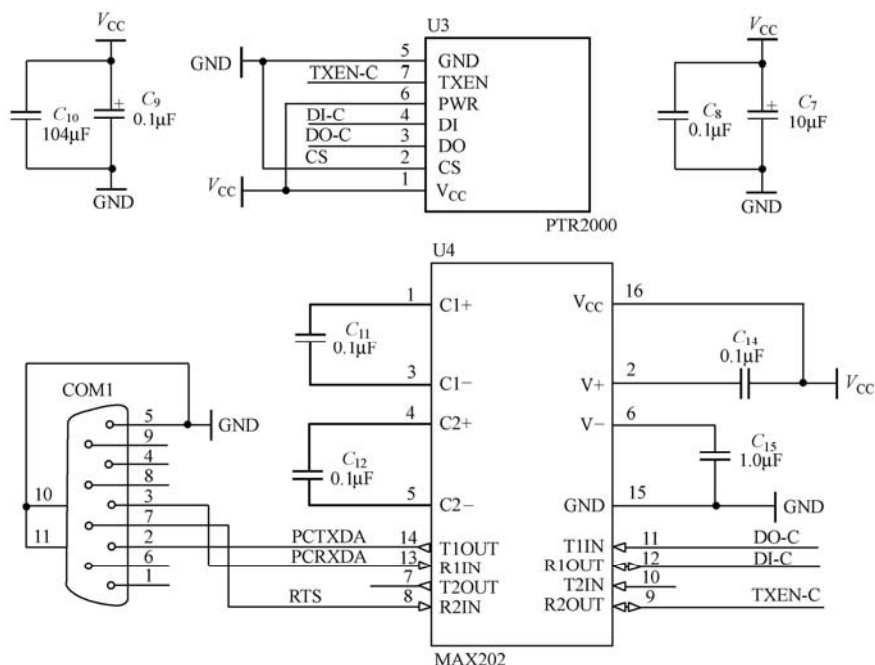


图 8-18 PC 和 PTR2000 接口电路原理图

8.2.4 软件设计

本例中由两个通信终端，分别是单片机和 PC，两部分的软件相互对应，设置各自的 PTR2000 模块的功能工作状态。这里着重介绍单片机端的软件设计。

出于方便的考虑，本例中已固定了通信频道，并且让 PTR2000 模块一直处于工作状态，不能使用待机状态（这两点已经在硬件的设计上有所体现）。这样，单片机和 PC 端软件就这要在于 PTR2000 发射还是接收状态的设置。

发射：收到接受命令后，应先将 PTR2000 模块置于发射模式，即将 TXEN 置 1。然后等待一段时间才可以发送数据（PTR2000 从接收模式转换成发送模式需要一个切换时间，大约 5 ms）。发送结束后，将模块置于接收状态，即将 TXEN 置 0。

接收：接收时应将 PTR2000 置于接收模式，即将 TXEN 置 0。单片机在不发送时的时候应尽量处于接收状态。

1. 程序流程图

本例的程序流程如图 8-19 所示。

单片机端默认的 PTR2000 状态为接收状态，通过串口中断来识别由 PC，即通过无线信道传输过来的指令，再根据具体指令完成对应的操作。需要注意的是，将 PTR2000 模块设置为发射状态的时候至少要等待 5 ms 的时间才可以发送；发送完毕后，向 PC 端发送“发送结束指令”，然后将 PTR2000 重新设置位接收状态。

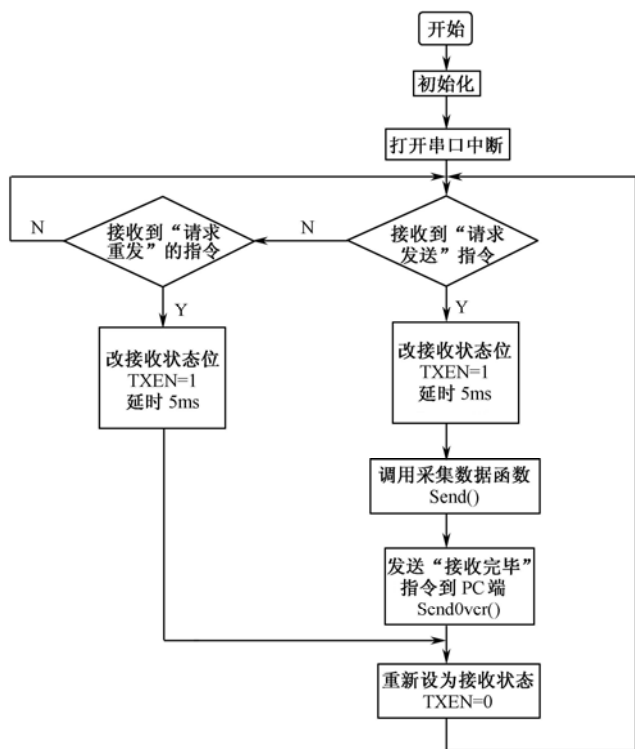


图 8-19 单片机端程序流程图

2. 程序代码

下面是程序的完整代码。

```

#include <reg52.h> // 引用标准库的头文件
#include <stdio.h>
#define uchar unsigned char
#define uint unsigned int

#define REQ_SEND          0x33
#define REQ_RESEND        0x66
#define SEND_OVER         0x99

//为简化起见，假设了 7 位固定的采集数据
#define DATA0            0x10
#define DATA1            0x20
#define DATA2            0x30
#define DATA3            0x40
#define DATA4            0x50
#define DATA5            0x60
#define DATA6            0x70
  
```



```

sbit TXEN = P2^0;
sbit CS = P2^1;
sbit PWR = P2^2;

void ClearT_buf();           //清除 t_buf 函数
void ClearR_buf();          //清除 r_buf 函数
void Delay();                //延时 5 ms

void GetData();              //采集数据
void Send();                 //采集并发送函数
void SendOver();             //通知 PC 发送结束函数

xdata uchar Flag_Start = 0 ; //开始接受数据标志位
xdata uchar Flag_RSend = 0 ; //请求发送标志位
xdata uchar Flag_ReSend = 0 ; //请求重发标志位
xdata uchar Flag_Delay = 1 ; //延时 5 ms 标志

xdata uchar Data[7];         //采集的 7 个字节数据

xdata uchar tCount = 0 ;
xdata uchar t_buf[11];       // 1 开始字节 "$", 1 长度字节 LEN,
                             // 7 B 数据, 1 校验和字节,
                             // 1 结束字节 "*"

xdata uchar rCount = 0 ;
xdata uchar r_buf[5];        // 1 开始字节 "$", 1 长度字节 LEN,
                             // 1 字节指令, 1 校验和字节,
                             // 1 结束字节 "*"

/* 定时器 0 中断服务子程序 */
void timer() interrupt 1 using 2
{
    Flag_Delay = 0;
    TH0 = -5000/256;
    TL0 = -5000%256;
}

//串口中断服务子程序
void serial ( ) interrupt 4 using 1
{
    RI = 0 ;

```

```

//判断是否收到字符 ‘$’，其数值为 0x24，置开始接收标志位
if ((!Flag_Start) && (SBUF == 0x24))
{
    Flag_Start = 1;
}

if (Flag_Start)
{
    if (rCount < 5)
    {
        r_buf[rCount] = SBUF;
        rCount ++;
    }
    //判断是否收到字符 ‘*’，其数值为 0x2a，根据接收的指令设置相应标志位
    if ((r_buf[rCount - 1] == 0x2a) || (rCount == 5))
    {
        rCount = 0;
        Flag_Start = 0;
        if (r_buf[2] == REQ_SEND)                //收到“请求发送”指令
        {
            Flag_RSend = 1;
        }
        if (r_buf[2] == REQ_RESEND)              //收到“请求重发”指令
        {
            Flag_ReSend = 1;
        }
    }
}
else
    ClearR_buf();
}
}
//清除 t_buf 函数
void ClearT_buf(void)
{
    uchar xdata i;
    for (i=0;i++;i<11)
    {
        t_buf[i] = 0;
    }
}
//清除 r_buf 函数
void ClearR_buf(void)

```

```

{
    uchar xdata i ;
    for (i=0;i++;i<5)
    {
        r_buf[i] = 0;
    }
}

//延时函数
void Delay(void)
{
    TR0=1;
    ET0=1;
    while( Flag_Delay);
    ET0 = 0;
    TR0 = 0;
    Flag_Delay = 1;
}

//采集数据函数经过简化处理，取固定的 7 B 数据
void GetData(void)
{
    Data[0]=DATA0;
    Data[1]=DATA1;
    Data[2]=DATA2;
    Data[3]=DATA3;
    Data[4]=DATA4;
    Data[5]=DATA5;
    Data[6]=DATA6;
}

//单片机端发送数据函数
void Send(void)
{
    uchar xdata j = 0;
    uchar xdata len = 0;
    uchar xdata CheckSum = 0 ;
    t_buf[0]=0x24;           //起始位
    t_buf[1]=0x07;           //7 B 数据
    len=t_buf[1];
    CheckSum = CheckSum + len;
    for ( j=0;j++;j<len)
    {

```

```

        t_buf[j+2] = Data[j];
        CheckSum = CheckSum + t_buf[j+2];
    }
    t_buf[9] = CheckSum;           //校验和字节
    t_buf[10] = 0x2A;             //停止位

    for (j=0;j++;j<11)
    {
        TI =0 ;
        SBUF = t_buf[j];
        while ( TI ==0 );
        TI =0 ;
    }
}

//通知 PC 端发送结束函数
void SendOver(void)
{
    TI =0 ;
    SBUF = 0x24;
    while ( TI ==0 );
    TI =0 ;
    SBUF = 0x01;
    while ( TI ==0 );
    TI =0 ;
    SBUF = SEND_OVER;             //通知 PC 端 “发送结束”
    while ( TI ==0 );
    TI =0 ;
    SBUF = 0x99;                  //校验字节
    while ( TI ==0 );
    TI =0 ;
    SBUF = 0x2A;
    while ( TI ==0 );
    TI =0 ;
}

void main(void)
{
    ClearT_buf();
    ClearR_buf();
    TXEN = 0;                     //初始为接收状态
    PWR = 1;                      //正常工作模式
}

```

```

CS = 0;                                //选择通道 0
EA=0;
//11.0592 MHz, 9 600 波特率, 工作方式 1, 8 数据位, 1 停止位, 无奇偶校验
TMOD = 0x21;
SCON = 0x50;                            //串行口工作方式 1, REN=1
PCON = 0x00;                            //SMOD=0

TL1 = 0xfd;
TH1 = 0xfd;
TR1=1;                                  //定时器 1 开始计时

PT0 = 1;                                //定时器 0 高优先级
TH0 = -5000/256;                         //5 ms
TL0 = -5000%256;

IE = 0x90;                              //EA=1, ES=1

while (1)
{
    if (Flag_RSend)                      // “请求发送” 指令
    {
        TXEN = 1;                       // 改变为发射状态
        Delay();                          // 延时 5 ms
        GetData();                       // 采集数据
        Send();                          // 发送
        Flag_RSend = 0;
        SendOver();                      // 通知 PC 发送结束
        TXEN = 0;                        // 重设为接收状态
    }
    else if (Flag_ReSend)                 // “请求重发” 指令
    {
        TXEN = 1;                       // 改变为发射状态
        Delay();                          // 延时 5 ms
        Send();                          // 重发
        Flag_ReSend = 0;
        SendOver();                      // 通知 PC 发送结束
        TXEN = 0;                        // 设为接收状态
    }
}
}

```

第9章 综合应用实例

本章将介绍一些 C51 单片机的综合应用实例，包括 I²C、GPS、USB、以太网等，这些实例具有很高的实际应用价值。

9.1 I²C总线接口技术在IC卡上的应用

随着 I²C 总线接口技术的成熟与发展，其应用也越来越广泛。目前，世界上许多公司都生产单片机，并对某种单片机进行再扩展。扩展单片机的方法包括并行总线和串行总线。因为串行总线连线少，结构简单，可以大大地简化系统硬件设计。

9.1.1 实例说明

通用存储器 IC 卡在各领域都得到了广泛的应用，这得益于它简单的结构和功能、低廉的生产成本并且使用方便。目前，用于 IC 卡的通用存储器芯片多为 E²PROM，其常用的协议主要有二线串行连接协议（IBC）和三线串行连接协议。I²C 总线的数据传送速率在标准工作方式下为 100 kb/s，在快速方式下，最高传送速率可达 400 kb/s。

9.1.2 I²C接口技术

I²C 接口技术是计算机的一种控制技术，属于二线式串行总线，各种被控制的电路都并联连接在这条总线上，每个电路和模块都有唯一的地址。在信息的传输过程中，根据不同的功能需要，I²C 总线上并接的每一模块电路既可以是主控器（或发送器），又可以是被控器（或接收器）。

1. I²C总线的基本结构

采用 I²C 总线标准的器件，内部要求具有 I²C 接口电路，而且内部各单元电路要按功能划分为若干相对独立的模块，并通过软件寻址实现片选，以减少器件片选线的连接。CPU 不仅能通过指令将某个功能单元电路挂靠或摘离总线，还可对该单元的工作状况进行检测，从而实现对硬件系统的扩展与控制。

I²C 总线接口电路结构如图 9-1 所示。

2. 双向传输的接口特性

通常的单片机串行接口的发送和接收是独立的，并各用一条线。当器件向总线上发送信息时，它就是发送器（也叫做主控器），而当器件从总线上接收信息时，它就是接收器（也叫做被控器）。因此，总线上主和从（即发送和接收）的关系不是固定的。主控器用于启动总线上传送数据并产生时钟以开放传送的器件，此时任何被寻址的器件均被认为是被控器。I²C 总线的控制完全由挂接在总线上的主控器送出的地址和数据决定，在总线上，既没有中心机，也没有优先级。

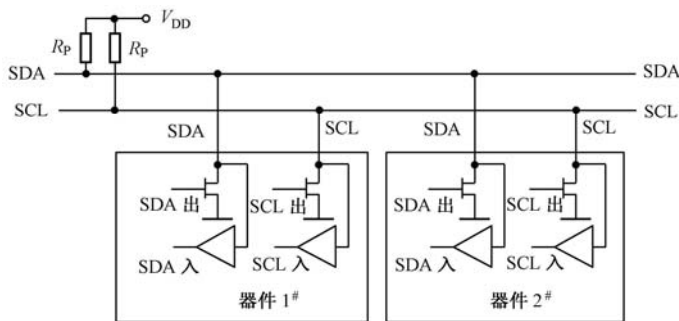


图 9-1 I²C 总线接口电路结构图

SDA 和 SCL 都是双向 I/O 线，通过上拉电阻接正电源。当总线空闲时，两根线都是高电平。连接总线的器件的输出级必须是集电极或漏极开路，以具有线“与”功能。

3. I²C 总线上的时钟信号

在 I²C 总线上传送信息时的时钟同步信号通过挂接在 SCL 时钟线上的所有器件的逻辑“与”完成，并会受 SCL 线从高电平到低电平的跳变影响，当某个器件的时钟信号变为低电平时，将使 SCL 线一直保持低电平，SCL 线上的所有器件开始进入低电平时。此时，低电平时短的器件的时钟由低电平至高电平的跳变也不能影响 SCL 线的状态，于是这些器件将进入高电平等待的状态。当所有器件的时钟信号都变为高电平时，低电平时结束，SCL 线被释放返回高电平，即所有的器件都同时开始进入高电平时。然后，第一个结束高电平时的器件又将 SCL 线拉成低电平。这样，就在 SCL 线上产生一个同步时钟。由此可见，时钟低电平时间由时钟低电平时最长的器件确定，而时钟高电平时间由时钟高电平时最短的器件确定。

4. 数据的传送

在数据传送时，必须确认数据传送的开始和结束条件，开始和结束条件的定义如图 9-2 中的 B 段和 C 段所示。

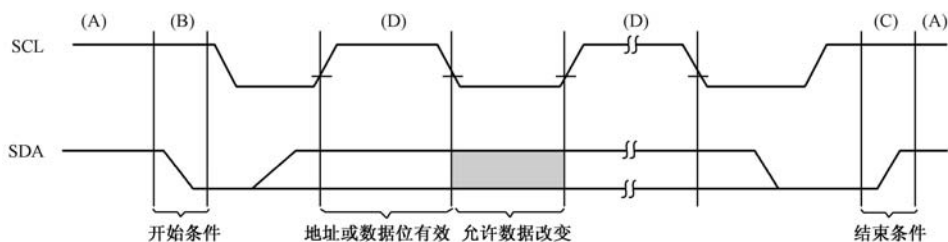


图 9-2 开始和结束条件

当时钟总线 SCL 为高电平时，数据线 SDA 由高电平跳变为低电平定义为“开始”条件；当 SCL 线为高电平时，SDA 线发生低电平到高电平的跳变为“结束”条件。开始和结束信号由主控器产生。在开始信号以后，总线被认为处于忙状态。在结束信号以后的一段时间内，总线被认为是空闲的。

总线空闲 (A)：数据线和时钟线同时为高电平。

启动数据传输 (B)：时钟 (SCL) 为高电平时，SDA 从高电平跳变为低电平表示起始

条件产生。起始条件必须先于所有的命令产生。

停止数据传输（C）：时钟（SCL）为高电平时，SDA 从低电平跳变为高电平表示停止条件产生。所有操作都必须以停止条件结束。

数据有效（D）：数据线的状态表明数据何时有效。在起始条件之后，数据线在时钟处于高电平期间保持稳定。所有操作必须在时钟信号为低电平期间改变数据线。一位数据位对应一个时钟脉冲，在起始条件和停止条件之间传输的数据字节数由主控器决定。

5. 总线竞争的仲裁

当一条总线上有多个器件时，可能会出现两个或多个主控器同时占用总线的情况，这种情况叫做总线竞争。 I^2C 总线具有多主控能力，可以对发生在 SDA 线上的总线竞争进行仲裁，仲裁的原则是：发生总线竞争时，如果一个主控器发送高电平，另一个主控器发送低电平，则发送电平与此 SDA 总线电平不符的那个器件将自动关闭其输出级；其次是地址位的比较，如果主控器寻址同一个被控器，则进入数据位的比较，从而确保了竞争仲裁的可靠性。利用 I^2C 总线上的信息进行仲裁，不会造成信息的丢失。

9.1.3 芯片 24LC01B的介绍及应用

24LC01B 是内含 128×8 位低功耗 CMOS 的 E^2PROM ，具有宽工作电压（2.5~5 V）、擦写次数多（大于 10 000 次）、写入速度快（小于 1 ms）等特点。

1. 24LC01B的引脚说明

图 9-3 是 24LC01B 芯片的引脚图。

A0、A1、A2 是三条地址线，用于确定芯片的硬件地址。 V_{CC} 为电源端， V_{SS} 为接地端，SDA 为串行数据输入/输出引脚，数据通过这条双向 I^2C 总线串行传送。SCL 为串行时钟输入线，WP 为写保护端。

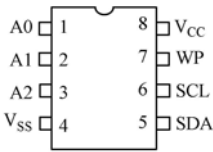


图 9-3 24LC01B 的引脚封装图

24LC01B 引脚的具体说明如表 9-1 所示。

表 9-1 24LC01B 引脚说明

引 脚 名 称	说 明
A0、A1、A2	芯片地址输入引脚
V_{SS}	电源接地
V_{CC}	标称条件下，如果 V_{CC} 低于 1.5 V，则 V_{CC} 阈值检测电路会禁止内部的擦写逻辑
SDA	串行数据引脚为双向引脚，用于把地址和数据输入/输出器件。该引脚为漏极开路。因此，SDA 总线要求在该引脚与 V_{CC} 端之间接入上拉电阻（通常频率为 100 kHz 时，该电阻阻值为 10 k Ω ，频率为 400 kHz 和 1 MHz 时，阻值为 2 k Ω ）。对于正常的数据传输，只允许在 SCL 为低电平期间改变 SDA 电平。而 SDA 电平在 SCL 高电平期间若发生变化，表明起始和停止条件产生
SCL	串行时钟：该输入引脚用于数据传输同步
WP	写保护引脚，该引脚必须连接到 V_{SS} 端或者 V_{CC} 端。如果连接到 V_{SS} 端，写操作使能，如果连接到 V_{CC} 端，写操作被禁止，但读操作不受影响

2. 24LC01B的总线特性

24LC01B 支持双向、二线数据传输协议。如果器件被定义为发送器，则该器件发送数

据到总线；定义为接收器，反之则接收来自总线的数据。总线由主控器控制，24LC01B 作为被控器。主控器提供串行时钟（SCL），控制总线访问和产生起始和停止条件，主控器决定采取何种工作模式，主控器和被控器都可以作为发送器或接收器。

总线协议定义如下：

- ① 只有在总线空闲时才可启动数据传输；
- ② 数据传输期间，当时钟线为高电平时，数据线都必须保持稳定，此时改变数据线将视为起始或停止条件；
- ③ 由于被寻址的接收器在接收到每一字节数据后都要发送一个确认位，主控器则应提供一个额外的时钟来传输确认位。在确认时钟脉冲内，器件确认必须拉低 SDA 线。在确认时钟的高电平期间，SDA 线以这种方式保持稳定的低电平。在读操作期间，主控器必须发送一个结束信号给被控器，而不是在被控器输出最后一个数据字节之后产生一个确认位，在这种情况下，被控器 24LC01B 将释放数据线为高电平，从而使主控器能够产生停止条件。确认信号的时序如图 9-4 所示。

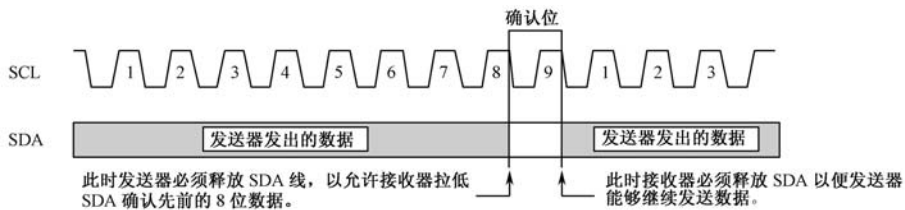


图 9-4 确认信号的时序

9.1.4 硬件设计

图 9-5 为 24LC01B 与 89C51 单片机进行串行数据输入/输出的电路原理图。

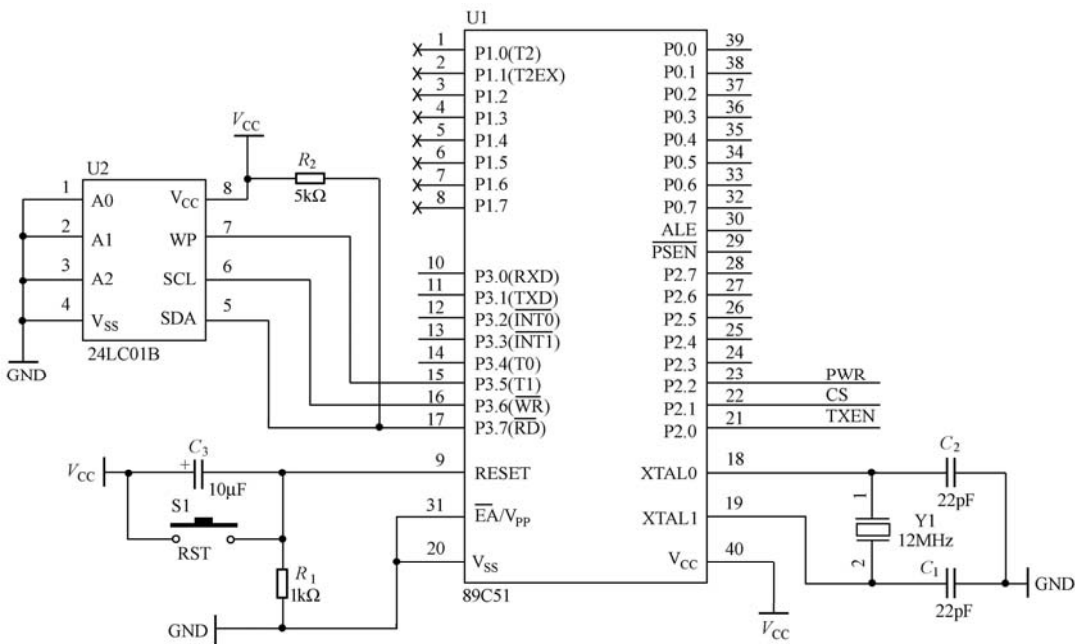


图 9-5 进行串行数据输入/输出的主要部分电路原理图

利用 89C51 的 P3.6 口和 P3.7 口分别作为 24LC01B 的串行时钟输入和串行数据输出，WP 端与 P3.5 口相连，配合单片机实现硬件写保护的功能。由于系统中只使用了这一个 E²PROM 芯片，故 A0~A2 接地。

只要 P3.6 口和 P3.7 口串行时钟和数据符合前面介绍的 I²C 总线的技术规范，即可实现对 24LC01B 的读/写。

9.1.5 软件设计

软件设计时要注意结合硬件电路图，它们之间是密切相关的。I²C 总线是通过 SDA（串行数据线）及 SCL（串行时钟线）两根线连接到总线上的器件进行传送信息的，程序中的各个函数实现的不同操作体现了这一点。

1. 程序流程图

定义 sbit SDA= P3^7，即 P3.7 口与 IC 卡的 SDA 引脚相连，定义 sbit SCL=P3^6，即 P3.6 口与 SCL 相连，定义 sbit WP=P3^5 即 P3.5 口与 WP 引脚相连。在程序中加入多个 _nop_() 是为了让 SDA 或者 SCL 上的信号达到稳定，_nop_() 的多少可以根据实际系统的稳定性而定。图 9-6 给出了程序的简易流程图。

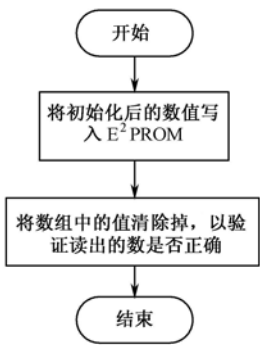


图 9-6 IC 卡程序的简易流程图

2. 程序代码

本例的程序代码说明如下：

```
//AT89C51 12MHz
#include <reg51.h>
#include <intrins.h>
#define uchar unsigned char
#define uint unsigned int

sbit SentData= P3^7;           //串行数据
sbit Clock= P3^6;             //串行时钟
sbit WP= P3^5;                //硬件写保护
//延时子函数
void delay(uchar b)
{
    uint a;
    for(;b>0;b--)
    {
        for(a=0;a<125;a--)
        {}
    }
}
```

```

}
}

//开始准备
void Start(void)
{
    SentData=1;
    Clock=1;
    _nop_();
    _nop_();
    _nop_();
    _nop_();
    SentData=0;
    _nop_();
    _nop_();
    _nop_();
    _nop_();
}

//停止条件
void Stop(void)
{
    SentData=0;
    Clock=1;
    _nop_();
    _nop_();
    _nop_();
    _nop_();
    SentData=1;
    _nop_();
    _nop_();
    _nop_();
    _nop_();
}

//应答
void Ack(void)
{
    SentData=0;
    _nop_();
    _nop_();
    _nop_();

```

```

_nop_();
Clock=1;
_nop_();
_nop_();
_nop_();
_nop_();
Clock=0;
}

```

//反相应答

void UnAck(void)

```

{
SentData=1;
_nop_();
_nop_();
_nop_();
_nop_();
Clock=1;
_nop_();
_nop_();
_nop_();
_nop_();
Clock=0;
}

```

//发送数据子程序，Data 为要求发送的数据

void Send(uchar Data)

```

{
uchar BC=8;                                //位数控制
uchar temp;                                //中间变量控制
do
{
temp=Data;
Clock=0;
_nop_();_nop_();_nop_();_nop_();
if((temp&0x80)==0x80)                        //如果最高位是 1
SentData=1;
else
SentData=0;
Clock=1;
temp=temp<<1;                                //RLC
}
}

```

```

Data=temp;
BC--;
}while(BC);
Clock=0;
}

```

//读一个字节的数据,并返回该字节值

```

uchar Read(void)

```

```

{
uchar temp=0;
uchar temp1=0;
uchar BC=8;
SentData=1;
do{
Clock=0;
_nop_();
_nop_();
_nop_();
_nop_();
Clock=1;
_nop_();
_nop_();
_nop_();
_nop_();
if(SentData)                                //如果 SDATA=1
temp=temp|0x01;                             //temp 的最低位置 1
else
temp=temp&0xfe;                             //否则 temp 的最低位清 0
if(BC-1)
{
temp1=temp<<1;
temp=temp1;
}
BC--;
}while(BC);
return(temp);
}

```

```

void WrToROM(uchar Data[],uchar Address,uchar Num)

```

```

{
uchar a;

```

```

uchar *PData;
PData=Data;
for(a=0;a<Num;a++)
{
    Start();                                //启动信号
    Send(0xa0);
    Ack();
    Send(Address+a);                        //地址
    Ack();
    Send(*(PData+a));
    Ack();
    Stop();
    delay(20);
}
}

```

```

void RdFromROM(uchar Data[],uchar Address,uchar Num)
{
    uchar b;
    uchar *PData;
    PData=Data;
    for(b=0;b<Num;b++)
    {
        Start();
        Send(0xa0);
        Ack();
        Send(Address+b);
        Ack();
        Start();
        Send(0xa1);
        Ack();
        *(PData+b)=Read();
        Clock=0;
        UnAck();
        Stop();
    }
}

//主程序
void main()
{
    uchar Nber[4]={1,2,3,4};

```

```

WP= 1;
WrToROM(Nber,4,4);           //将初始化后的数值写入 E2PROM
delay(20);
Nber[0]=0;
Nber[1]=0;
Nber[2]=0;
Nber[3]=0;                   //将数组中的值清掉，验证读出数正确与否
RdFromROM(Nber,4,4);
}

```

9.2 C51 单片机实现GPS定位设计

GPS (global positioning system)，即全球定位系统，是一个由覆盖全球的 24 颗卫星组成的卫星系统，它可以保证在任意时刻，地球上任意一点都可以同时观测到 4 颗卫星，保证卫星可以采集到该观测点的经纬度和高度，用以实现导航、定位、授时等功能。

9.2.1 实例效果说明

本例要实现通过单片机获取 GPS 定位信息，即用单片机控制 GPS 器件的数据读取和传输过程。

9.2.2 GPS的介绍

全球定位系统 (GPS) 是 20 世纪 70 年代由美国陆海空三军联合研制的新一代空间卫星导航定位系统。GPS 全球卫星定位系统由三部分组成：空间部分——GPS 星座、地面控制部分——地面监控系统，以及用户设备部分——GPS 信号接收机。

GPS 导航系统测量出已知位置的卫星到用户接收机之间的距离，然后综合多颗卫星的数据就可知道接收机的具体位置。卫星的位置可以根据星载时钟所记录的时间在卫星星历中查出。通过记录卫星信号传播到用户所经历的时间乘以光速得到用户到卫星的距离，由于大气层电离层的干扰，这一距离并非真实距离，而是伪距 (PR)。当 GPS 卫星正常工作时，会不断地用 1 和 0 二进制码元组成的伪随机码 (简称为伪码) 发射导航电文。导航电文中的内容包括遥测码、转换码和第 1、2、3 数据块，其中最重要的则为星历数据。当用户接收到导航电文时，提取出卫星时间并将其与自己的时钟进行对比便可得知卫星与用户的距离，再利用导航电文中的卫星星历数据便可推算出卫星发射电文时所处位置，用户在 WGS-84 大地坐标系中的位置速度等信息便可得知。

由于用户接收机使用的时钟与卫星星载时钟不能总是同步，所以除了用户的三维坐标 x 、 y 、 z 外，还要引进一个 Δt (卫星与接收机之间的时间差) 作为未知数，然后用 4 个方程将这 4 个未知数解出来。所以至少要能接收到 4 个卫星的信号才能知道用户和卫星之间的距离。

按定位方式的不同，GPS 定位分为单点定位和相对定位（差分定位）。单点定位就是根据一台接收机的观测数据来确定接收机位置的方式，它只能采用伪距观测量，例如车船等的概略导航定位。相对定位（差分定位）是根据两台以上接收机的观测数据来确定观测点之间的相对位置的方法，它既可采用伪距观测量也可采用相位观测量，例如大地测量。

GPS 观测量会受到很多因素的影响，包含了卫星和接收机的时差、大气传播延迟、多路径效应等误差，还会受到卫星广播星历误差的影响，所以，在精度要求高，而且接收机之间距离较远时，应选用双频接收机，因为双频接收机可以根据两个频率的观测量抵消大气中电离层误差的主要部分。

9.2.3 GARMIN GPS 25LP介绍

本例中所选用的 GPS 器件是 GARMIN GPS 25LP，它的特点是定位速度快、工作稳定、耐电压冲击和高抗干扰性。

1. GARMIN GPS 25LP的特性

GARMIN GPS 25LP 的特性如下：

- 并行 12 通道瞬间锁定可视卫星；
- 长寿命后备锂电使重捕速度更快；
- 全屏蔽封装具备优秀抗电磁干扰特点；
- 1×10^{-12} s 脉冲输出精度可达到 $\pm 1 \mu\text{s}$ ；
- 标准 NMEA0183 语句可选择输出；
- 二进制格式输出和 Motorola 格式兼容；
- 多种供电模式、电平输出模式可供选择；
- 输出电压 3.6~6 V (LV_x ，低压模式)；6~40 V (HV_x ，高压模式)；
- 电平输出 RS-232 HVS LVS；CMOS LVC HVC；
- 差分精度可达 5 m 天线和 GPS25LP 的完美结合。

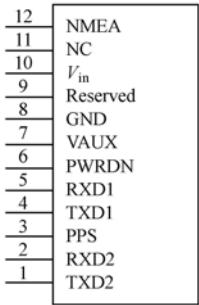
其接收性能如下：

- 定位时间：重新捕获 < 2 s；
- 自动搜索：90 s；
- 热启动：15 s；冷启动：45 s；
- 更新率：1~1/900 s 可调；
- 速度精度：0.1 m/s；
- 速度限制：515 m/s；
- 坐标系统：102 个预定义，1 个自定义；
- 加速度限制：6 g。

输入/输出特性如下：

- 电气特性：两个 RS-232 兼容串行口；
- CMOS 通讯速率：300 b/s、600 b/s、1 200 b/s、2 400 b/s、4 800 b/s、9 600 b/s、19 200 b/s 可选；

- 数据格式：NEMA V2.0 ASCII/二进制，可设置；
- 输入数据：初始位置/日期/时间，2D/3D 方式坐标系统，RTCM-104 差分校正；
- 输出数据：速度、时间、机器/卫星状态、几何因子及误差估计；
- 秒脉冲输出： 1×10^{-12} s 精度 $\pm 1 \mu\text{s}$ 。



2. GARMIN GPS 25LP的引脚说明

GARMIN GPS 25LP 的接口如图 9-7 所示。

GARMIN GPS 25LP 接口说明如表 9-2 所示。

图 9-7 GARMIN GPS 25LP 的接口

表 9-2 GARMIN GPS 25LP 接口说明

引 脚	名 称	描 述
1	串口 2（出）	相位数据输出
2	串口 2（入）	接收 RTCM SC-104 版本 2.1 的 GPS 差分信息
3	秒脉冲	上升沿与 GPS 秒同步，电压升降时间 300 ns，阻抗 250 Ω。开路输出电压为：低电压 0 V，高电压 V_{in} 。高电平持续时间从 20~980 ms 可调。接 50 Ω负载后输出 700 mV（峰-峰）信号。对于在 50%电压点测得的秒脉冲时间，接 50 Ω负载后将比空载提前 50 ns
4	串口 1（出）	异步串行数据输出。RS-232 电平，提供 NMEA 0183 版本 2.0 的数据，从 300~19 200 b/s 可选，默认值为 4 800 b/s
5	串口 1（入）	异步串行数据输入。RS-232 电平，最大输入电压范围为-25~25 V。该输入口也可以直接与有 RS-232 极性的标准的 3~5 V 的 CMOS 逻辑电平连接，要求低电压小于 0.8 V，高电压大于 2.4 V。最大负载阻抗是 4.7 kΩ。该口主要用于接收对 OEM 板的初始化信息和配置信息
6	外部关机	激活后将关闭内部整流器，并将供电电流降低到 20 mA 以下。高于 2.7 V 激活，低于 0.5 V 则不激活
7	备用电源	为内部电池充电。输入电压为直流 4~35 V
8	地	电源地和信号地
9	电源	与 10 引脚相连接
10	电源	电压 3.6~6 V。内部有 6.8 V 的稳压管和热敏电阻，但出现瞬变电流和过压的现象时，将关闭接收机直到供电恢复正常
11	保留	留待扩展
12	NMEA	提供 CMOS 电平的 NMEA 0183 语句输出，输出与 4 引脚相同

9.2.4 硬件设计

电路原理如图 9-8 所示。

图中，单片机工作在 12 MHz 时钟下，它和 GARMIN GPS 25LP 的接口只有两根串口线 GPS_TXD 和 GPS_RXD。

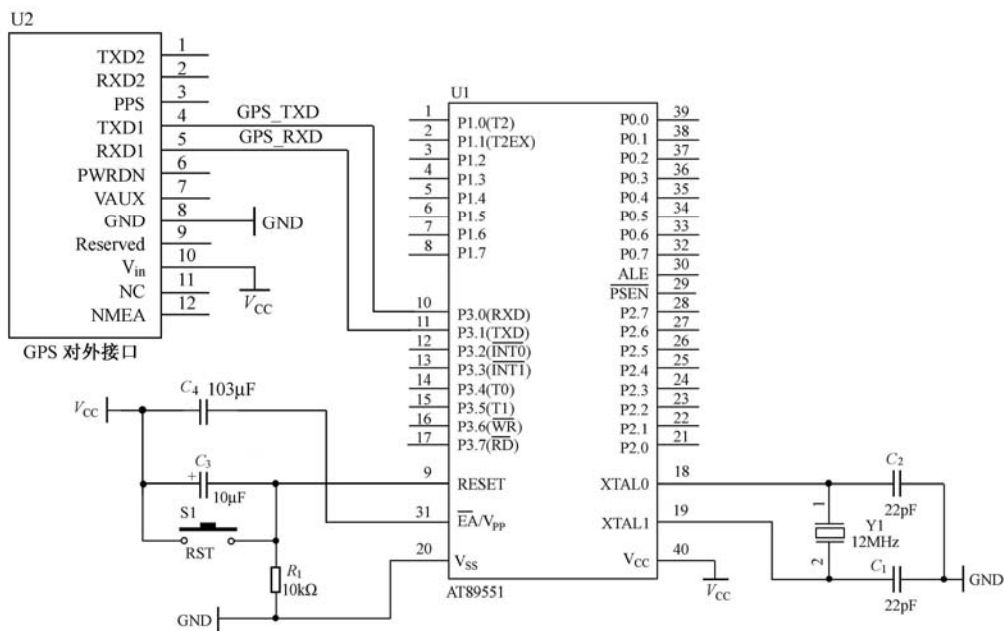


图 9-8 单片机实现 GPS 定位电路原理图

9.2.5 软件设计

本例中在进行软件设计前首先要对 GARMIN GPS 25LP 特有的输入/输出语句格式有所了解。GARMIN 的 GPS OEM 板所输出的数据是以美国国家海洋电子协会（National Marine Electronics Association）的 NMEA 0183 ASCII 码接口协议为基础的。这里我们仅介绍输入指令中的接收机配置信息（PGRMC）。其他的语句介绍可以参考 ARMIN GPS 25LP 的技术资料。

1. 接收机配置信息

PGRMC 语句可以配置 GPS 接收机的工作状态。配置参数将被保存在永久性存储器中，在下次加电时将会自动生效。如果该语句中没有错误，接收机将会有有一个自动回复，如果接收机检测到该语句中有错误，回复的 PGRMC 语句将为当前值。向 GPS 接收机发送 PGRMCE 命令也可以获得当前的 PGRMC 值。

如表 9-3 为接收机配置信息（PGRMC）。

表 9-3 接收机配置信息（PGRMC）

配置位	说 明
<1>	定位模式，A=自动，2=仅为 2D 定位（必须提供高度数值），3=仅为 3D 定位
<2>	相对于平均海平面的海拔高度（-1 500.0~18 000.0 m）
<3>	坐标系统索引。如果使用自定义坐标系统，下面从<4>到<8>的字段就必须包含有效的数值。
<4>	自定义坐标系，半长轴，6 360 000.000~6 380 000.000 m（分辨率 0.001 m）
<5>	自定义坐标系，扁率倒数，285.0~310.0（分辨率 10^{-9} ）

配置位	说 明
<6>	自定义坐标系，DX，-5 000.0~5 000.0 m（分辨率 1 m）
<7>	自定义坐标系，DY，-5 000.0~5 000.0 m（分辨率 1 m）
<8>	自定义坐标系，DZ，-5 000.0~5 000.0 m（分辨率 1 m）
<9>	差分模式，A=自动，D=仅为差分
<10>	NMEA 0183 的波特率，1=1 200 b/s，2=2 400 b/s，3=4 800 b/s，4=9 600 b/s，5=19 200 b/s，6=300 b/s，7=600 b/s，8=38 400 b/s
<11>	速度滤波器，0=不滤波，1=自动滤波，2~255=滤波时间常数（s）
<12>	秒脉冲模式，1=无秒脉冲输出，2=1 Hz
<13>	秒脉冲长度，0~48=(n+1)×20 ms
<14>	递推时间，1~30 s

说明：\$GPRMC,<1>,<2>,<3>,<4>,<5>,<6>,<7>,<8>,<9>,<10>,<11>,<12>,<13>,<14>*hh<CR><LF>

2. 程序代码

本例的程序代码说明如下：

```
#include <reg52.h>                                // 引用标准库的头文件
#include <stdio.h>
#include <absacc.h>

#define uchar unsigned char
#define uint unsigned int

uchar StrGps_LONG[8];                               // 存放经度数据
uint i;
uint Rec_Flag;                                       // 记录标志
uint IGps_LONG;                                     // 经度数据计数变量
uint Comma_Num;                                     // 逗号计数变量

void DisAllOut();                                   // 配置 GPS 模块禁用所有输出语句子函数
void EnGPRMC();                                     // 配置 GPS 模块使能$GPRMC 输出语句子函数

void main()
{
    //串口初始化
    TMOD = 0x20;
    TL1= 0xfd;
    TH1 = 0xfd;
    SCON = 0x40;                                     // 方式 1：10 位异步收发，波特率由定时器控制，REN=0
    PCON = 0x00;                                     // SMOD = 0
```

```

IE = 0x90;           // EA = 1, ES = 1
TR1 = 1;             // 定时器 1 启动

DisAllOut();
EnGPRMC();

delay(1000);          // 延时 1 000 ms

REN = 1;              // 开始接收数据

while(1);
}

//配置 GPS 模块禁用所有输出语句函数
void DisAllOut()
{

    TI = 0;
    SBUF = 0x24;       // 发送 “$”
    while(!TI);
    TI = 0;
    SBUF = 'P';         // 发送 “P”
    while(!TI);
    TI = 0;
    SBUF = 'G';         // 发送 “G”
    while(!TI);
    TI = 0;
    SBUF = 'R';         // 发送 “R”
    while(!TI);
    TI = 0;
    SBUF = 'M';         // 发送 “M”
    while(!TI);
    TI = 0;
    SBUF = 'O';         // 发送 “O”
    while(!TI);
    TI = 0;
    SBUF = ',';         // 发送 “,”
    while(!TI);
    TI = 0;
    SBUF = ',';         // 发送 “,”
    while(!TI);

```

```

    TI = 0;
    SBUF = '2';                // 发送 “2”
    while(!TI);
    TI = 0;
    SBUF = 0x2a;               // 发送 “*”
    while(!TI);
    TI = 0;
}

```

//配置 GPS 模块使能\$GPRMC 输出语句

void EnGPRMC()

```

{
    TI = 0;
    SBUF = 0x24;               // 发送 “$”
    while(!TI);
    TI = 0;
    SBUF = 'P';                // 发送 “P”
    while(!TI);
    TI = 0;
    SBUF = 'G';                // 发送 “G”
    while(!TI);
    TI = 0;
    SBUF = 'R';                // 发送 “R”
    while(!TI);
    TI = 0;
    SBUF = 'M';                // 发送 “M”
    while(!TI);
    TI = 0;
    SBUF = 'O';                // 发送 “O”
    while(!TI);
    TI = 0;
    SBUF = ',';                // 发送 “,”
    while(!TI);
    TI = 0;
    SBUF = 'G';                // 发送 “G”
    while(!TI);
    TI = 0;
    SBUF = 'P';                // 发送 “P”
    while(!TI);
    TI = 0;
}

```

```

    SBUF = 'R';                                // 发送 “R”
    while(!TI);
    TI = 0;
    SBUF = 'M';                                // 发送 “M”
    while(!TI);
    TI = 0;
    SBUF = 'C';                                // 发送 “C”
    while(!TI);
    TI = 0;
    SBUF = ',';                                // 发送 “,”
    while(!TI);
    TI = 0;
    SBUF = '1';                                // 发送 “1”
    while(!TI);
    TI = 0;
    SBUF = 0x2a;                                // 发送 “*”
    while(!TI);
    TI = 0;
}

```

//延时 t ms

void delay(uint t)

```

{
    uint i;
    while(t-->0)
    {
        for (i=0;i<125;i++)
            {}
    }
}

```

//串口接收中断

serial() interrupt 4 using 1

```

{
    RI = 0;                                // 清除中断标志位
}

```

//判断是否收到 GPRMC 格式语句的第一个字符 “\$”，收到后开始记录数据，并设置记录

//标志 Rec_Flag

```

    if (SBUF == 0x24)
    {
        Rec_Flag = 1;
    }
}

```

```

        i = 0;
        IGps_LONG = 0;
        Comma_Num = 0;
    }

    //处理 GPRMC 中的数据信息
    if (Rec_Flag == 1)
    {
        if(SBUF == 0x2c)
            Comma_Num++;

        //第 5 个逗号后的字符属于经度数据
        if (Comma_Num == 3)
        {
            StrGps_LONG[IGps_LONG] = SBUF; // 存入字符串 StrGps
            IGps_LONG++;
        }

        //判断是否收到 GPRMC 格式语句的字符 “*”，收到则结束记录
        if (SBUF == 0x2a)
        {
            StrGps_LONG[IGps_LONG] = '\0'; // 字符串的结束符
            Rec_Flag = 0;
            IGps_LONG = 0;
            Comma_Num = 0;

            REN = 0;
            delay(500); // 延时 500 ms
            REN = 1;
        }
    }
}

```

9.3 USB总线接口设计

USB 作为一种新型的接口技术，以其简单易用、速度快等特点而备受青睐。

本例简单介绍了 USB 接口的特点和 PHILIPS 公司的 USB 接口芯片 PDIUSB12，以及它是如何和单片机连接的。

9.3.1 实例说明

早期的计算机系统常用串口或并口连接外围设备，每个接口都需要占用计算机的系统

资源（如中断，I/O 地址，DMA 通道等）。因为串口和并口都是点对点的连接，一个接口仅支持一个设备，因此每添加一个新的设备，就需要添加一个 ISA/EISA 或 PCI 卡来支持，同时系统还需要重新启动才能驱动新的设备。

USB 总线是一种串行总线标准，主要用于 PC 与外围设备的互连。USB 总线具有低成本、使用简单、支持即插即用、易于扩展等特点，已被广泛地用在 PC 及嵌入式系统上。

本例为 PDIUSBD12 与单片机的接口电路设计，着重介绍 USB 标准，USB 接口芯片 PDIUSBD12 以及 PDIUSBD12 与单片机的硬件连接。

9.3.2 USB 简介

USB 英文全称是 universal serial bus，翻译成中文就是“通用串行总线”。它不是一种新的总线标准，而是应用在 PC 领域的接口技术。USB 用一个 4 针插头作为标准插头，采用菊花链形式把所有的外设连接起来，最多可以连接 127 个外部设备，并不损失带宽。USB 工作需要主机硬件、操作系统和外设三个方面的支持。USB 接口可以通过专门的 USB 连机线实现双机互连，并可以通过 Hub 扩展出更多的接口。USB 具有传输速度快（USB1.1 是 12 Mb/s，USB2.0 是 480 Mb/s）、使用方便、支持热插拔、连接灵活、独立供电等优点，可以连接鼠标、键盘、打印机、扫描仪、摄像头、闪存盘、MP3 机、手机、数码相机、移动硬盘、外置光软驱、USB 网卡、ADSL Modem、Cable Modem 等几乎所有的外部设备。

1. USB 的历史

USB 总线是在 1994 年由 Intel、DEC、Microsoft、IBM 等公司联合提出的。

USB 的版本：

第一代：USB 1.0/1.1 的最大传输速率为 12 Mb/s，于 1996 年推出。

第二代：USB 2.0 的最大传输速率高达 480 Mb/s，USB 1.0/1.1 与 USB 2.0 的接口是相互兼容的。

第三代：USB 3.0 在理论上最大传输速率可达 5 Gb/s，向下兼容 USB 1.0/1.1/2.0。

从 1994 年 11 月 11 日发表了 USB V0.7 版本以后，到现在已经发展为 2.0 版本，成为计算机中的标准扩展接口。目前主板中主要是采用 USB1.1 和 USB2.0，各 USB 版本间能很好地兼容。其中，USB2.0 最为普及，USB2.0 有高速、全速和低速三种工作速度，高速的最大传输速率是 480 Mb/s，全速的最大传输速率是 12 Mb/s，低速的最大传输速率是 1.5 Mb/s，它们的实际传输速率分别在 25~400 Mb/s、500 kb/s~10 Mb/s 和 10~100 kb/s 之间。

2. USB 的优点

USB 的优点如下：

（1）使用简单

接口一致，连线简单。系统可对设备进行自动检测和配置，支持热插拔。不需要重新启动就可以添加新的设备系统。

（2）应用范围广

数据报文附加信息少，带宽利用率高，可同时支持同步传输和异步传输两种传输方式。

(3) 纠错能力强

实时地管理设备插拔。在 USB 协议中包含了传输错误管理、错误恢复等功能，还可以根据不同的传输类型来处理传输错误。

(4) 总线供电

USB 总线可为连接设备提供 5 V/100 mA 的供电，最大可提供 500 mA 的电流。其本身也可采用自供电方式。

(5) 低成本

USB 接口电路简单，易于实现，特别是低速设备。USB 系统接口电缆也比较简单，成本比串口/并口低。

9.3.3 USB接口芯片PDIUSBD12 介绍

PDIUSBD12 通常用于基于微控制器的系统并与微控制器通过高速通用并行接口进行通信，支持本地 DMA。传输该器件采用模块化的方法实现一个 USB 接口，允许在可用的微控制器中选择最合适的作为系统微控制器，允许使用现存的体系结构并使固件投资降低到最小。减少了开发时间、风险和成本。PDIUSBD12 完全符合 USB1.1 协议规范，能适应大多数设备类规范的设计。因此，PDIUSBD12 适合用做很多外围设备，如打印机、扫描仪和数码相机等，并可以降低成本。

PDIUSBD12 挂起时的低功耗以及 LazyClock 输出符合 ACPI、OnNOW 和 USB 电源管理设备的要求。低功耗工作允许实现总线供电的外围设备。

PDIUSBD12 集成了例如 SoftConnect、GoodLink、可编程时钟输出、低频晶振和终端电阻等功能。这些功能都能在系统实现时节省成本，并能实现更高级的 USB 功能。

1. PDIUSBD12 的产品性能

- 符合 USB 1.1 协议规范；
- 集成了 SIE FIFO 存储器/收发器和电压调整器的高性能 USB 接口芯片；
- 适用于大多数设备类规范的设计；
- 与任何微控制器/微处理器有高速 2 Mb/s 的并行接口；
- 全自动 DMA 操作；
- 集成了 320 B 的多配置 FIFO 存储器；
- 主端点有双缓存配置，增加吞吐量，容易实现实时数据传输；
- 在块传输模式下有 1 Mb/s 的数据传输速率，在同步传输模式下有 1 Mb/s 的数据传输速率；
- 具有总线供电能力，有很好的 EMI 性能；
- 在挂起时有可控制的 LazyClock 输出；
- 可通过软件控制 USB 总线连接 SoftConnect；
- 在 USB 传输时有闪亮的 USB 连接指示灯 GoodLink；
- 时钟频率输出可编程；

- 符合 ACPI、OnNOW 和 USB 电源管理要求；
- 具有内部上电复位和低电压复位电路 ；
- 有 SO - 18 和 TSSOP - 28 封装；
- -40℃~+85℃工业级温度；
- 片内 8 kV 静电保护；
- 双电压工作（3.3±0.3 V）或扩大的 5 V 电压范围（3.6~5.5 V）；
- 多中断模式，方便块传输和同步传输。

2. PDIUSB12 的引脚说明及内部结构

PDIUSB12 的引脚如图 9-9 所示。

PDIUSB12 的引脚说明如表 9-4 所示。

PDIUSB12 的内部结构如图 9-10 所示。

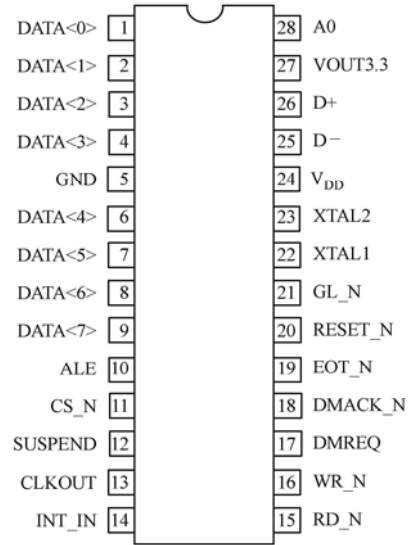


图 9-9 PDIUSB12 的引脚图

表 9-4 PDIUSB12 的引脚说明

引 脚 名 称	引 脚 说 明
DATA<0>~DATA<7>	8 位双向数据
GND	地
ALE	地址锁存
CS_N	片选，低电平有效
SUSPEND	进入挂起状态
CLKOUT	可编程时钟输出
INT_N	中断输出，低电平有效
RD_N	读选通，低电平有效
WR_N	写选通，低电平有效
DMREQ	DMA 请求
DMACK_N	DMA 响应，低电平有效
EOT_N	DMA 传输结束，低电平有效
RESET_N	复位，低电平有效、异步
GL_N	GoodLink 发光二极管指示器，低电平有效
XTAL1	晶振连接 1（6 MHz）
XTAL2	晶振连接 2（6 MHz）
V _{DD}	正电源 4.0~5.5 V
D-	USB D-数据线
D+	USB D+数据线
VOUT3.3	3.3 V 输出
A0	地址位。A0=1，选择命令；A0=0 选择数据

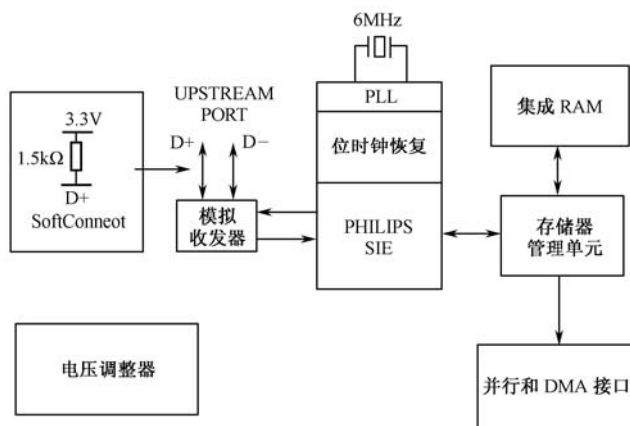


图 9-10 PDIUSB12 的内部结构图

PDIUSB12 内部结构包括模拟收发器、电压调整器、PLL、位时钟恢复、GoodLink、存储器管理单元 MMU 和集成 RAM、并行和 DMA 接口等。

3. PDIUSB12 的时序图及经典应用

PDIUSB12 的并行接口时序如图 9-11 所示。

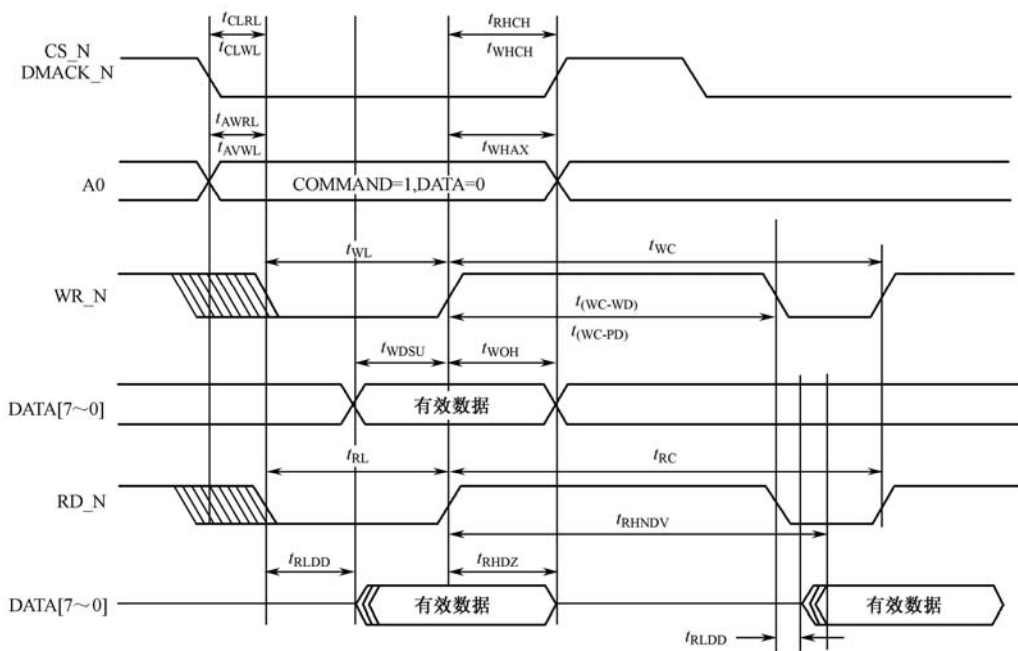


图 9-11 PDIUSB12 的并行接口时序图

PDIUSB12 与单片机的常用电路如图 9-12 所示。

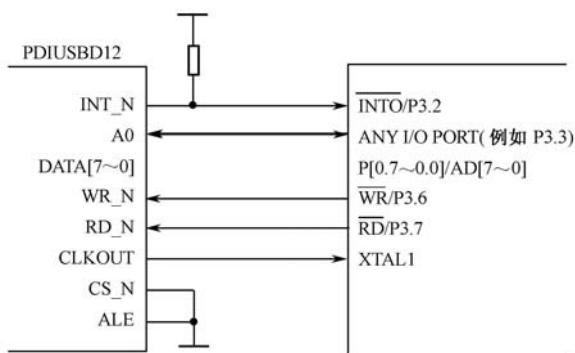


图 9-12 PDIUSBD12 与单片机的常用电路图

9.3.4 硬件设计

本例分为 PDIUSBD12 的外围电路和 PDIUSBD12 与单片机接口电路两个部分。

1. PDIUSBD12 的外围电路

PDIUSBD12 的外围电路如图 9-13 所示。

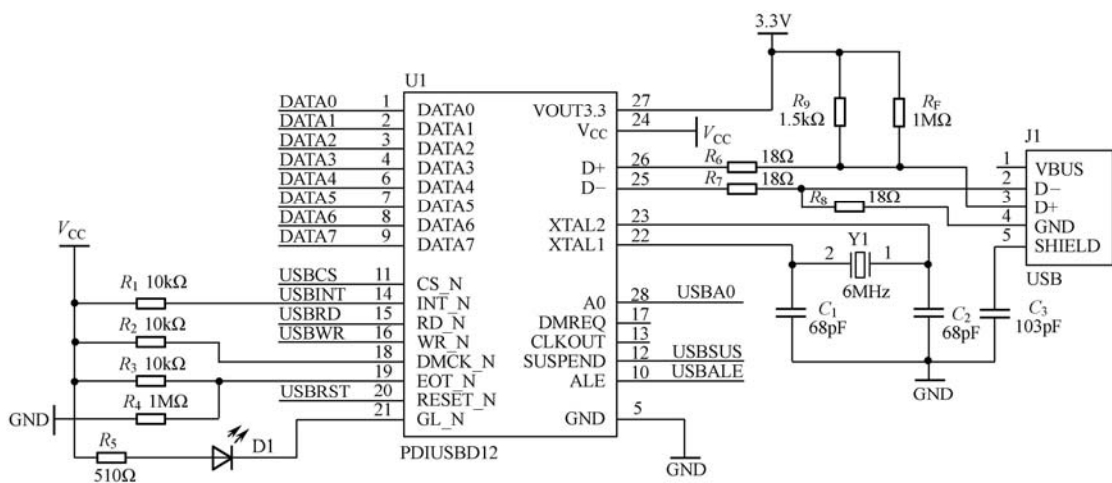


图 9-13 PDIUSBD12 的外围电路图

2. PDIUSBD12 与单片机接口电路

PDIUSBD12 与单片机接口电路如图 9-14 所示。

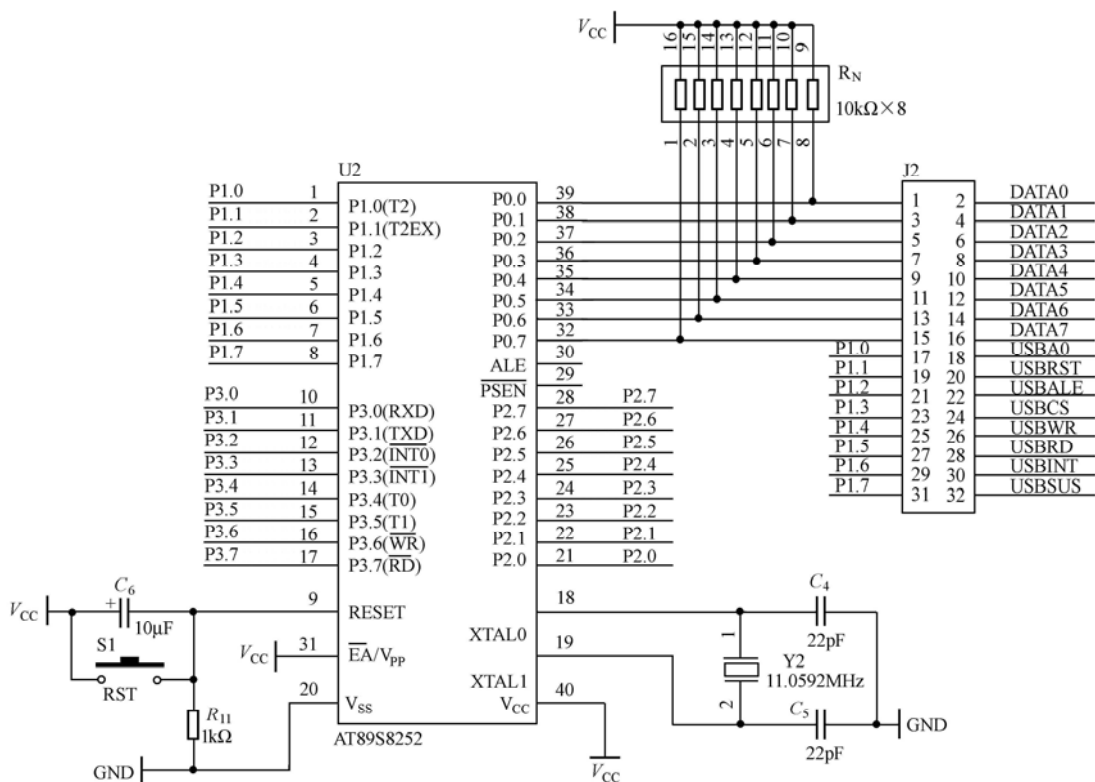


图 9-14 PDIUSBD12 与单片机接口电路图

9.3.5 软件设计

程序的源代码如下：

```
#include<reg51.h>
sbit SUSPEND = P3^5;

#define D12_INT_ENDP0OUT  0x0001           // 中断寄存器位定义
#define D12_INT_ENDP0IN   0x0002
#define D12_INT_ENDP1OUT  0x0004
#define D12_INT_ENDP1IN   0x0008
#define D12_INT_ENDP2OUT  0x0010
#define D12_INT_ENDP2IN   0x0020
#define D12_INT_BUSRESET  0x0040
#define D12_INT_SUSPENDCHANGE 0x0080
#define D12_INT_EOT       0x0100

#define D12_SETUPPACKET   0x20           //读最后处理状态寄存器的设置信息包 0010 0000b
#define EP0_PACKET_SIZE   16             // p0 最大 16 B
#define USB_ENDPOINT_DIRECTION_MASK 0x80 //设备请求类型,传输方向 D7 1000 0000b
```

```

#define USB_REQUEST_TYPE_MASK      0x30      // bmRequest 的设置
#define USB_REQUEST_MASK            0x0f
#define USB_STANDARD_REQUEST       0x00      // 5,6 位的定义
#define USB_VENDOR_REQUEST         0x20
#define USB_DEVICE_DESCRIPTOR_TYPE 0x01
// 描述符类型 设备描述符 01H, 配置描述符 02, 接口描述符 04H, 端点描述符 05H
#define USB_CONFIGURATION_DESCRIPTOR_TYPE 0x02
#define CONFIG_DESCRIPTOR_LENGTH    0x002E      //配置描述符总长度

#define SWAP(x)    (((x) & 0x00FF) << 8) | (((x) >> 8) & 0x00ff)      //交换高低 8 位
#define MSB(x)     (((x) >> 8) & 0x00ff)      // 取数据高 8 位
#define LSB(x)     ((x) & 0x00ff)             // 取数据低 8 位

typedef union _Event_Flags_          // 定义 USB 事件标志数据类型
{
    struct _Bit_Flags_
    {
        unsigned char Timer           : 1;      //定时器溢出事件标记
        unsigned char BusReset        : 1;      //USB 总线复位标志
        unsigned char Suspend         : 1;      //USB 器件挂起标志
        unsigned char SetupPacket     : 1;      //收到 SETUP 包标志
        unsigned char RemoteWakeup    : 1;      //远程唤醒标志
        unsigned char InISR           : 1;      //USB 中断服务标志
        unsigned char ControlState    : 2;      //控制端点处理状态
        // 0:IDEL      空闲状态
        // 1:TRANSMIT 数据发送状态
        // 2:RECEIVE 数据接受状态

        unsigned char Configuration  : 1;      // 配置标志 (0:未配置;1:已配置)
        unsigned char Port1RxDone    : 1;      // 端口 1 收到数据标志
        unsigned char Port2RxDone    : 1;      // 端口 2 收到数据标志
        unsigned char Port1TxFull    : 1;      // 端口 1 输出缓冲区满标志
        unsigned char Port2TxFull    : 1;      // 端口 2 输出缓冲区满标志

        unsigned char Reserve        : 3;      // 保留, 未使用
    }Bits;
    unsigned short int Value;
}EVENT_FLAGS;          // 事件标志数据类型

typedef struct _DEVICE_REQUEST_
{

```

```

    unsigned char bmRequestType;           // 请求类型
    unsigned char bRequest;                // USB 请求
    unsigned short wValue;                  // USB 请求值
    unsigned short wIndex;                  // USB 请求索引
    unsigned short wLength;                 // 计数长度
}DEVICE_REQUEST;

#define MAX_CONTROLDATA_SIZE 8
typedef struct _control_xfer
{
    DEVICE_REQUEST DeviceRequest;           // USB 请求结构体
    unsigned short wLength;                  // 传输数据的总字节数
    unsigned short wCount;                   // 传输字节数统计
    unsigned char * pData;                   // 传输数据指针
    unsigned char dataBuffer[MAX_CONTROLDATA_SIZE]; // 请求的数据
}CONTROL_XFER;

static EVENT_FLAGS EventFlags;             //定义为全局变量,用于与主程序的通信
static CONTROL_XFER ControlData;           //保存 SETUP 包请求类型和请求数据
unsigned char idata EndPoint1Buffer[4];    //控制端点缓存
unsigned char idata EndPoint2Buffer[64];   //主端点缓存
//硬件提取层,多路地址/数据总线方式读/写函数
void Outportb(unsigned int Addr, unsigned char Data)
{
    *((unsigned char xdata *) Addr) = Data;
}

unsigned char Inportb(unsigned int Addr)
{
    return *((unsigned char xdata *) Addr);
}

void USB_Delay1ms(unsigned int count)
{
    unsigned int i,j;
    for(i=0;i<count;i++)
        for(j=0;j<120;j++);
}

#define D12_DATA    0
#define D12_COMMAND 1

```

//PDIUSB D12 命令接口函数

void D12_SetMode(unsigned char bConfig,unsigned char bClkDiv)

```
{
    Outportb(D12_COMMAND,0xF3);
    Outportb(D12_DATA,bConfig);
    Outportb(D12_DATA,bClkDiv);
}
```

//设置端点函数

void D12_SetEndpointStatus(unsigned char bEndp,unsigned char bStalled)

```
{
    Outportb(D12_COMMAND,0x40+bEndp);
    Outportb(D12_DATA,bStalled);
}
```

//应答函数

void D12_AcknowledgeEndpoint(unsigned char endp)

```
{
    Outportb(D12_COMMAND,endp);
    Outportb(D12_COMMAND,0xF1);
    if(endp==0)
        Outportb(D12_COMMAND,0xF2);
}
```

//设置地址使能

void D12_SetAddressEnable(unsigned char bAddress,unsigned char bEnable)

```
{
    Outportb(D12_COMMAND,0xD0);
    if(bEnable) bAddress |= 0x80;
    Outportb(D12_DATA,bAddress);
}
```

//设置端点使能

void D12_SetEndpointEnable(unsigned char bEnable)

```
{
    Outportb(D12_COMMAND,0xD8);
    if(bEnable)
        Outportb(D12_DATA,1);
    else
        Outportb(D12_DATA,0);
}
```

//读中断寄存器

unsigned short D12_ReadInterruptRegister(void)

```
{
    unsigned char b1;
```



```

    unsigned int j;
    Outportb(D12_COMMAND,0xF4);
    b1=Inportb(D12_DATA);
    j=Inportb(D12_DATA);
    j<<=8;
    j+=b1;
    return j;
}
//读取端点状态
unsigned char D12_ReadEndpointStatus(unsigned char EndPoint)
{
    unsigned char BackValue;

    if(EventFlags.Bits.InISR == 0)
        EA = 0;

    Outportb(D12_COMMAND, 0x80 + EndPoint);        //读取端点状态
    BackValue = Inportb(D12_DATA);

    if(EventFlags.Bits.InISR == 0)
        EA = 1;

    return BackValue;
}
//读端点最后处理状态
unsigned char D12_ReadLastTransactionStatus(unsigned char bEndp)
{
    Outportb(D12_COMMAND,0x40+bEndp);
    return Inportb(D12_DATA);
}
//读端口函数
unsigned char D12_ReadEndpoint(unsigned char endp,unsigned char len,unsigned char *buf)
{
    unsigned char i,j;
    Outportb(D12_COMMAND,endp);
    if((Inportb(D12_DATA)&0xff)==0)/* define D12_FULLEEMPTY as 0xFF by newer
        return 0;
    Outportb(D12_COMMAND,0x80+endp);
    i=Inportb(D12_DATA);
    i=i&0x60;

```

```

Outportb(D12_COMMAND,0xF0);
j=Inportb(D12_DATA);
j=Inportb(D12_DATA);
if(j>len)
    j=len;
for(i=0;i<j;i++)
    *(buf+i)=Inportb(D12_DATA);
Outportb(D12_COMMAND,0xF2);
return j;
}

```

```

unsigned char D12_ReadEndpoint_Int(unsigned char endp,unsigned char len,unsigned char *buf)
{
    unsigned char i,j;
    Outportb(D12_COMMAND,endp);
    if((Inportb(D12_DATA)&0xff)==0)/* define D12_FULLEEMPTY as 0xFF by newer
        return 0;
    Outportb(D12_COMMAND,0x80+endp);
    i=Inportb(D12_DATA);
    i=i&0x60;

```

```

Outportb(D12_COMMAND,0xF0);
j=Inportb(D12_DATA);
j=Inportb(D12_DATA);
if(j>len)
    j=len;
for(i=0;i<j;i++)
    *(buf+i)=Inportb(D12_DATA);
Outportb(D12_COMMAND,0xF2);
return j;
}

```

```

unsigned char D12_WriteEndpoint(unsigned char endp,unsigned char len,unsigned char * buf)
{
    unsigned char i;
    Outportb(D12_COMMAND,endp);
    Inportb(D12_DATA);
    Outportb(D12_COMMAND,0xF0);
    Outportb(D12_DATA,0);

```

```

    Outportb(D12_DATA,len);
    for(i=0;i<len;i++)
        Outportb(D12_DATA,*(buf+i));
    Outportb(D12_COMMAND,0xFA);
    return len;
}

```

```

unsigned char D12_WriteEndpoint_Int(unsigned char endp,unsigned char len,unsigned char * buf)
{
    unsigned char i;
    Outportb(D12_COMMAND,endp);
    Inportb(D12_DATA);
    Outportb(D12_COMMAND,0xF0);
    Outportb(D12_DATA,0);
    Outportb(D12_DATA,len);
    for(i=0;i<len;i++)
        Outportb(D12_DATA,*(buf+i));
    Outportb(D12_COMMAND,0xFA);
    return len;
}

```

```

void DisconnectUSB(void)
{
    D12_SetMode(0x02,0x03);//SET TO ONE? by newer
}

```

```

void InitialUSBInt(void);
void ConnectUSB(void)
{
    EventFlags.Value = 0x0000;
    InitialUSBInt();
    D12_SetMode(0x16,0x03);
}

```

```

void ReconnectUSB(void)
{
    SUSPEND = 0;
    DisconnectUSB();
    USB_Delay1ms(1000);
    ConnectUSB();
}

```

//中断服务程序

void InitialUSBInt(void)

```
{
    IT1=0; //低电平中断触发
    EX1=1; //允许外部中断
    PX1=0; //优先级低
    EA =1;
}
```

void EP0_Out(void)

```
{
    unsigned char ep_last,i;
    ep_last=D12_ReadLastTransactionStatus(0);//interrupt symbol
    if(ep_last&D12_SETUPPACKET)
    { //recieved SETUP packet  ---by newer
        ControlData.wLength=0;
        ControlData.wCount=0;
        if(D12_ReadEndpoint_Int(0,sizeof(ControlData.DeviceRequest),(unsignedchar*) (&(Control-
            Data.DeviceRequest)))!=sizeof(DEVICE_REQUEST))
        {
            D12_SetEndpointStatus(0,1);
            D12_SetEndpointStatus(1,1);
            EventFlags.Bits.ControlState=0; //should define USB_IDLE first --by newer
            return;
        }
    }
```

```
    ControlData.DeviceRequest.wValue=SWAP(ControlData.DeviceRequest.wValue);
    ControlData.DeviceRequest.wIndex=SWAP(ControlData.DeviceRequest.wIndex);
    ControlData.DeviceRequest.wLength=SWAP(ControlData.DeviceRequest.wLength);
```

```
D12_AcknowledgeEndpoint(0);
D12_AcknowledgeEndpoint(1);
ControlData.wLength=ControlData.DeviceRequest.wLength;
ControlData.wCount=0;
if(ControlData.DeviceRequest.bmRequestType&(unsigned char)
    USB_ENDPOINT_DIRECTION_MASK)
{
    EventFlags.Bits.SetupPacket=1; //recv from host? --by newer
    EventFlags.Bits.ControlState=1; //by newer
}
```

```

else
{
    if(ControlData.DeviceRequest.wLength==0)
    {
        EventFlags.Bits.SetupPacket=1;
        EventFlags.Bits.ControlState=0; //by newer
    }
    else
    {
        if(ControlData.DeviceRequest.wLength>16)//最大传 16 B
        {
            EventFlags.Bits.ControlState=0; //by newer
            D12_SetEndpointStatus(0,1);
            D12_SetEndpointStatus(1,1);
        }
        else
        {
            EventFlags.Bits.ControlState=2; //by newer
        }
    }
}
else if(EventFlags.Bits.ControlState==2)
{
    i=D12_ReadEndpoint_Int(0,EP0_PACKET_SIZE,ControlData.dataBuffer+ControlData.wCount);
    ControlData.wCount+=i;
    if(i!=EP0_PACKET_SIZE||ControlData.wCount>=ControlData.wLength)
    {
        EventFlags.Bits.SetupPacket=1;
        EventFlags.Bits.ControlState=0;
    }
}
else
    EventFlags.Bits.ControlState=0;
}

void EP0_In(void)
{
    short i=ControlData.wLength-ControlData.wCount;
    D12_ReadLastTransactionStatus(1);
    if(EventFlags.Bits.ControlState!=1) return;

```

```

if(i>=EP0_PACKET_SIZE)
{
    D12_WriteEndpoint_Int(1,EP0_PACKET_SIZE,ControlData.pData+ControlData.wCount);
    ControlData.wCount+=EP0_PACKET_SIZE;
    EventFlags.Bits.ControlState=1;
    return;
}
if(i!=0)
{
    D12_WriteEndpoint_Int(1,i,ControlData.pData+ControlData.wCount);
    ControlData.wCount+=i;
    EventFlags.Bits.ControlState=0;
    return;
}
D12_WriteEndpoint_Int(1,0,0);
EventFlags.Bits.ControlState=0;
}

void EP1_Out(void)
{
    unsigned char Length;
    D12_ReadLastTransactionStatus(2); // Clear interrupt flag
    Length = D12_ReadEndpoint_Int(2, sizeof(EndPoint1Buffer),EndPoint1Buffer);
    if(Length != 0)
        EventFlags.Bits.Port1RxDone = 1;
}

void EP1_In(void)
{
    D12_ReadLastTransactionStatus(3);
}

void EP2_Out(void)
{
    unsigned char Length,EP2Status;
    D12_ReadLastTransactionStatus(4); //Clear interrupt flag

    EP2Status = D12_ReadEndpointStatus(4);
    EP2Status&=0x60;
    Length = D12_ReadEndpoint(4,sizeof(EndPoint2Buffer),EndPoint2Buffer);
    if(EP2Status==0x60)
        Length = D12_ReadEndpoint(4,sizeof(EndPoint2Buffer),EndPoint2Buffer);

```

```

    if(Length != 0)
        EventFlags.Bits.Port2RxDone = 1;
}

void EP2_In(void)
{
    D12_ReadLastTransactionStatus(5);           //Clear interrupt flag
}
//请求处理
typedef struct _usb_device_descriptor
{
    unsigned char bLength;
    unsigned char bDescriptorType;
    unsigned int bcdUSB;
    unsigned char bDeviceClass;
    unsigned char bDeviceSubClass;
    unsigned char bDeviceProtocol;
    unsigned char bMaxPacketSize0;
    unsigned int idVendor;
    unsigned int idProduct;
    unsigned int bcdDevice;
    unsigned char iManufacturer;
    unsigned char iProduct;
    unsigned char iSerialNumber;
    unsigned char bNumConfigurations;
}USB_DEVICE_DESCRIPTOR;

code USB_DEVICE_DESCRIPTOR DeviceDescr=
{
    sizeof(USB_DEVICE_DESCRIPTOR),
    0x01,//USB_DEVICE_DESCRIPTOR_TYPE,
    SWAP(0x0100),
    0xDC,//USB_CLASS_CODE_TEST_CLASS_DEVICE,
    0, 0,
    EP0_PACKET_SIZE,
    SWAP(0x0471),
    SWAP(0x0666),
    SWAP(0x0100),
    0, 0, 0,
    25
};

```

//配置描述符

```
typedef struct _usb_configuration_descriptor
{
    unsigned char bLength[0x2e];
}USB_CONFIGURATION_DESCRIPTOR;
```

code USB_CONFIGURATION_DESCRIPTOR ConfigDescr=

```
{
    0x09,0x02,0x2e,0x00,0x01,0x01,0x00,0xa0,0x32,
    0x09,0x04,0x00,0x00,0x04,0xdc,0xa0,0xb0,0x00,
    0x07,0x05,0x81,0x03,0x02,0x00,0x0a,
    0x07,0x05,0x01,0x03,0x02,0x00,0x0a,
    0x07,0x05,0x82,0x02,0x40,0x00,0x0a,
    0x07,0x05,0x02,0x02,0x40,0x00,0x0a
};
```

//code_transmit

void code_transmit(unsigned char code *pRomData,unsigned short len)

```
{
    ControlData.wCount=0;
    if(ControlData.wLength>len)
        ControlData.wLength=len;
    ControlData.pData=pRomData;
    if(ControlData.wLength>=EP0_PACKET_SIZE)
    {
        D12_WriteEndpoint(1,EP0_PACKET_SIZE,ControlData.pData);
        ControlData.wCount+=EP0_PACKET_SIZE;
        EA = 0;
        EventFlags.Bits.ControlState=1;
        EA = 1;
    }
    else
    {
        D12_WriteEndpoint(1,ControlData.wLength,pRomData);
        ControlData.wCount+=ControlData.wLength;
        EA = 0;
        EventFlags.Bits.ControlState=0;
        EA = 1;
    }
}
```

//获取描述符

void get_descriptor(void)


```

{
    if(MSB(ControlData.DeviceRequest.wValue)==USB_DEVICE_DESCRIPTOR_TYPE)
    {
code_transmit((unsigned char code*)&DeviceDescr,sizeof(USB_DEVICE_DESCRIPTOR));
        return;
    }
    if(MSB(ControlData.DeviceRequest.wValue)==USB_CONFIGURATION_DESCRIPTOR_TYPE)
    {
        if(ControlData.DeviceRequest.wLength>CONFIG_DESCRIPTOR_LENGTH)
        {
            ControlData.DeviceRequest.wLength=CONFIG_DESCRIPTOR_LENGTH;
            //标识符总大小 2EH 字节，第二次请求时 wlength=0x00ff
        }
        //这里的 ConfigDescr 其实应该包括其他标识符!
code_transmit((unsigned char code*)&ConfigDescr,ControlData.DeviceRequest.wLength);
        return;
    }
}
//串口发送
void single_transmit(unsigned char *buf,unsigned char len)
{
    if(len<=EP0_PACKET_SIZE)
    {
        D12_WriteEndpoint(1,len,buf);
    }
}
//设置地址
void set_address(void)
{
    D12_SetAddressEnable((unsigned char)(ControlData.DeviceRequest.wValue&0xff),1);
//比如 wValue 是 “02 00” 应该得到 02
    single_transmit(0,0);
}

//设置配置
void set_configuration(void)
{
    if(ControlData.DeviceRequest.wValue==0)
    {
        single_transmit(0,0);
        EventFlags.Bits.Configuration=0;
    }
}

```

```

    D12_SetEndpointEnable(0);
}
else if(ControlData.DeviceRequest.wValue==1)
{
    single_transmit(0,0);
    D12_SetEndpointEnable(0);
    D12_SetEndpointEnable(1);
    EventFlags.Bits.Configuration=1;
}
}
//读取配置
void get_configuration(void)
{
    unsigned char c=EventFlags.Bits.Configuration;
    single_transmit(&c,1);
}
//读取设备接口信息
void get_interface(void)
{
    unsigned char txdat=0;
    single_transmit(&txdat,1);
}

static code void (*StandardDeviceRequest[])(void)=
{
    0,0,0,0,
    0,set_address,get_descriptor,0,
    get_configuration,set_configuration,get_interface,0,
    0,0,0,0
};

static code void (*VendorDeviceRequest[])(void)=
{
    0,0,0,0,
    0,0,0,0,
    0,0,0,0,
    0,0,0,0
};

void ControlHandler(void)
{

```

```

unsigned char type, req;
type=ControlData.DeviceRequest.bmRequestType&USB_REQUEST_TYPE_MASK;//0011,0000b
req=ControlData.DeviceRequest.bRequest&USB_REQUEST_MASK;//0000,1111b
if(type==USB_STANDARD_REQUEST)
    (*StandardDeviceRequest[req])();
else if(type==USB_VENDOR_REQUEST)
    (*VendorDeviceRequest[req])();
}

```

```

void USB_ISR(void) interrupt 2
{
    unsigned int i_st;
    EA = 0;
    EventFlags.Bits.InISR=1;
    i_st=D12_ReadInterruptRegister();
    if(i_st!=0)
    {
        if(i_st&D12_INT_ENDP0OUT)
            EP0_Out();
        if(i_st&D12_INT_ENDP0IN)
            EP0_In();
        if(i_st&D12_INT_ENDP1OUT)
            EP1_Out();
        if(i_st&D12_INT_ENDP1IN)
            EP1_In();
        if(i_st&D12_INT_ENDP2OUT)
            EP2_Out();
        if(i_st&D12_INT_ENDP2IN)
            EP2_In();
    }
    EventFlags.Bits.InISR=0;
    EA = 1;
}

```

```

extern EVENT_FLAGS EventFlags; //事件信号
extern unsigned char idata EndPoint1Buffer[4];

```

```

main()
{
    ReconnectUSB();
}

```

```

for(;;)
{
    if(EventFlags.Bits.SetupPacket)
    {
        EA = 0;
        EventFlags.Bits.SetupPacket = 0;
        ControlHandler();
        EA = 1;
    }
    if(EventFlags.Bits.Timer)
    {
        EventFlags.Bits.Timer = 0;
    }
    if(EventFlags.Bits.Port1RxDone)
    {
        EventFlags.Bits.Port1RxDone = 0;
    }
}
}

```

9.4 基于RTL8019AS的以太网接口实验

随着 Internet 在中国的普及，针对 Internet 的各种技术也随之产生。以太网是目前使用最广泛的网络标准。由于其不仅在市场上有着最低的 NIC（网络接口卡）和 Hub 端口价格，还具有维护简单、易于扩充等优点，它已经成为最受欢迎的技术。以太网协议已经广泛地应用在各种计算机网络，例如办公局域网、工业控制网络、因特网等。基于以太网协议的各种设备也不断出现，目前比较常用的 10 Mb/s 嵌入式控制芯片有 RTL8019AS、CS8900、DM9008 等。利用廉价的 C51 单片机和以太网控制芯片来实现以太网通信具有十分重要的意义。

本例是单片机控制以太网驱动控制芯片 RTL8019AS 实现以太网接口的一个实例。

9.4.1 实例说明

单片机具有体积小、功能强大等特点，是设计各种智能化仪器和控制设备的主要手段。随着网络技术和控制技术的蓬勃发展，各种智能仪器和设备连在一起可以组成网络。

本实例主要介绍了基于 C51 单片机的以太网卡硬件接口电路的设计，以及网卡驱动程序的开发。核心处理器采用 AT89C52 单片机，以太网卡控制芯片采用 RTL8019AS 芯片，将 TCP/IP 协议栈嵌入到单片机中，根据应用的需要将单片机中的 TCP/IP 协议进行简化，实现网络通信。

9.4.2 设计思路分析

本例的功能模块可以划分为以下几部分。

① 单片机系统，为整个电路的核心处理器部分，作用是对以太网接口的初始化以及以太网数据的发送和接收控制。

② 以太网接口电路，由以太网接口芯片 RTL8019AS 和 C51 单片机的接口电路组成，通过标准的网络接口与以太网连接。

③ C51 程序，包括以太网接口芯片初始化程序，以及发送数据和接收数据程序。

9.4.3 以太网协议

由于以太网在市场上有着最低的 NIC（网络接口卡）和 Hub 端口价格，还具有维护简单、易于扩充等优点，它已经成为最受欢迎的技术，也成为最广泛使用的网络标准。

1. CSMA/CD 技术

以太网使用 CSMA/CD（载波监听多路访问及冲突检测技术）技术，并以 10Mb/s 的速率运行在多种类型的电缆上。CSMA/CD 定义了以太网节点为传输数据如何获得对网络媒体的访问。

2. 以太网帧格式

以太网的帧是数据链路层的封装，网络层的数据包被加上帧头和帧尾就成为了可以被数据链路层识别的数据帧（成帧）。虽然帧头和帧尾所用的字节数是固定不变的，但依被封装的数据包大小的不同，以太网的帧长度也在变化，其范围是 64~1 514 B（8 B 的前导字不包含在其中）。

3. 工作模式

以太网卡的工作模式有半双工模式和全双工模式两种。

半双工传输模式可以实现以太网载波监听多路访问冲突检测，在同一时间只能传输单方向的数据。传统的共享 LAN 是在半双工模式下工作的，两个方向的数据同时传输时会产生冲突，这会降低以太网的效率。

全双工传输模式是采用点对点连接，因为它们使用双绞线中两个独立的线路，所以没有冲突。在全双工模式下，冲突检测电路不可用，所以每个全双工连接只用一个端口，用于点对点连接。标准以太网的传输效率可达到 50%~60% 的带宽，全双工模式在两个方向上都提供 100% 的效率。

4. 工作过程

当以太网中的一台主机要传输数据时，它的工作过程如下：

① 侦听信道上是否有信号在传输。如果有的话，表明信道处于忙状态，就继续侦听，直到信道空闲为止。

- ② 若没有侦听到任何信号，就开始传输数据。
- ③ 传输的时候继续侦听，如发现冲突则执行退避算法，随机等待一段时间后，重新执行步骤①。
- ④ 若未发现冲突则发送成功，计算机在试图再一次发送数据之前，必须在最近一次发送数据后等待 9.6 μs（以 10 Mb/s 的速率运行）。

5. 以太网协议

以太网协议有两种，一种是 IEEE 802.2/IEEE 802.3，另外一种是以以太网的封装格式。现在的操作系统均能同时支持这两种类型的协议格式。因此对我们来说只需要了解其中的一种就够了，特别是对单片机来说，不可能支持太多的协议格式。本例只介绍第二种格式的用于 10Mb/s 的以太网协议。

一个标准的以太网物理传输帧由 7 个部分组成，如表 9-5 所示。

表 9-5 以太网的物理传输帧结构表

PR	SD	DA	SA	TYPE	DATA	PCS
同步位	分隔位	目的地址	源地址	类型字段	数据段	帧校验序列
56 位	8 位	48 位	48 位	16 位	不超过 1 500 B	32 位
7 B	1 B	6 B	6 B	2 B	46~1 500 B	4 B

数据段的长度为 46~1 500 B，其他部分的长度都是固定不变的。以太网规定整个传输包的最大长度不能超过 1 514 B（14 B 为 DA、SA、TYPE），最小不能小于 60 B。除去 DA、SA、TYPE 共 14 B，还必须传输 46 B 的数据，当数据段的数据不足 46 B 时，则需要填充字符，填充字符的个数不包括在长度字段里；当超过 1 500 B 时，则需拆成多个帧进行传送。所有数据位的传输都是由从低位开始（但传输的位流是用曼彻斯特编码的）的，发送数据时，PR、SD、FCS 以及填充字段这几个数据段均由以太网控制器自动产生；而接收数据时，PR、SD 被跳过，控制器一旦检测到有效的前序字段（PR、SD），就认为接收数据开始。

以太网的物理传输帧结构位所表示的意义如下：

- ① PR：同步位，用于收发双方的时钟同步，同时也指明了传输的速率（10 Mb/s 和 100 Mb/s 的时钟频率不一样，所以 100 Mb/s 网卡可以兼容 10 Mb/s 网卡）是 56 位的二进制数“101010101010...”。
- ② SD：分隔位，表示下面跟着的是真正的数据，而不是同步时钟，为 8 位的 10101011，与同步位不同的是，SD 的最后 2 位是 11 而不是 10。
- ③ DA：目的地址，以太网的地址为 48 位（6 B）二进制地址，表明该帧传输给哪个网卡。如果为 FFFFFFFFFFFFFF，则是广播地址，广播地址的数据可以被任何网卡接收到。
- ④ SA：源地址，48 位，表明发送端的网卡地址，源地址同样是 6 B。
- ⑤ TYPE：类型字段，表明该帧的数据类型，不同协议的类型字段不同。
- ⑥ DATA：数据段，该段数据不能超过 1 500 B。

⑦ PAD: 填充位, 当数据段的数据不足 46 个字节时, 后面补 0 (也可以补其他值)。

⑧ FCS: 32 位数据校验位, 为 32 位的 CRC 校验, 该校验由网卡自动计算、自动生成、自动校验、自动在数据段后面填入。

以太网卡可以接收三种地址的数据: 广播地址; 多播地址和本身的地址。但网卡也可以设置为接收任何数据包 (用于网络分析和监控)。任何两个网卡的物理地址都是不一样的, 网卡地址由专门机构分配。不同厂家使用不同地址段, 同一厂家的任何一个网卡的地址都是唯一的。

9.4.4 芯片概述

本例采用的是由我国台湾 Realtek 公司生产的 RTL8019AS 高度集成以太网控制器, 它能够简单地实现即插即用 NE2000 兼容适配器, 这种适配器具有二重和功率下降的特性。通过三电平控制特性, RTL8019AS 是现有对网络设备 GREEN PC 理想的选择。为了特殊的应用而得到即插即用功能的兼容性, RTL8019AS 支持 JUMPER 和 JUMPERLESS 选项, 为了提供完全解决即插即用方案, RTL8019AS 集成了 10BASET 收发器, BNC 和 AUI 接口之间的自动检测功能。

RTL8019AS 以太网控制器的功能特性如下。

- 支持对 10BaseT 拓扑结构的自动极性修正;
- 兼容 Ethernet II 与 IEEE 802.3 (10Base5、10Base2、10BaseT) 标准;
- 全双工, 收发可同时达到 10 Mb/s 的速率;
- 内置 16 KB 的 SRAM, 用于收发缓冲, 降低对主处理器的速度要求;
- 支持 8/16 位数据总线, 8 个中断申请线以及 16 个 I/O 基地址选择;
- 支持 UTP, AUI 和 BNC 自动检测 (仅 RTL8019AS 支持);
- 支持 8 条 IRQ 总线、16 条 I/O 基本地址选项和额外 I/O 地址输入输出完全解码方式;
- 支持 16 KB, 32KB 和 64KB 的 BROM 和闪存接口;
- 支持存储器瞬时读/写 (仅 RTL8019AS 支持);
- 16 KB 的 SRAM (仅 RTL8019AS 支持);
- 使用 9346 (64×16 位 E²PROM) 存储资源配置和 ID 参数;
- 允许 4 个诊断 LED 引脚可编程输出;
- 免跳线方式, 网卡的 I/O 和中断由外接的 9346 的内容决定;
- 100 引脚的 PQFP 封装, 缩小了 PCB 尺寸。

1. 芯片的引脚分布内部结构

在介绍单片机的以太网卡硬件接口电路的设计前, 首先了解 RTL8019AS 以太网控制芯片的基础知识。

(1) 引脚分布

RTL8019AS 的引脚分布图如图 9-15 所示。

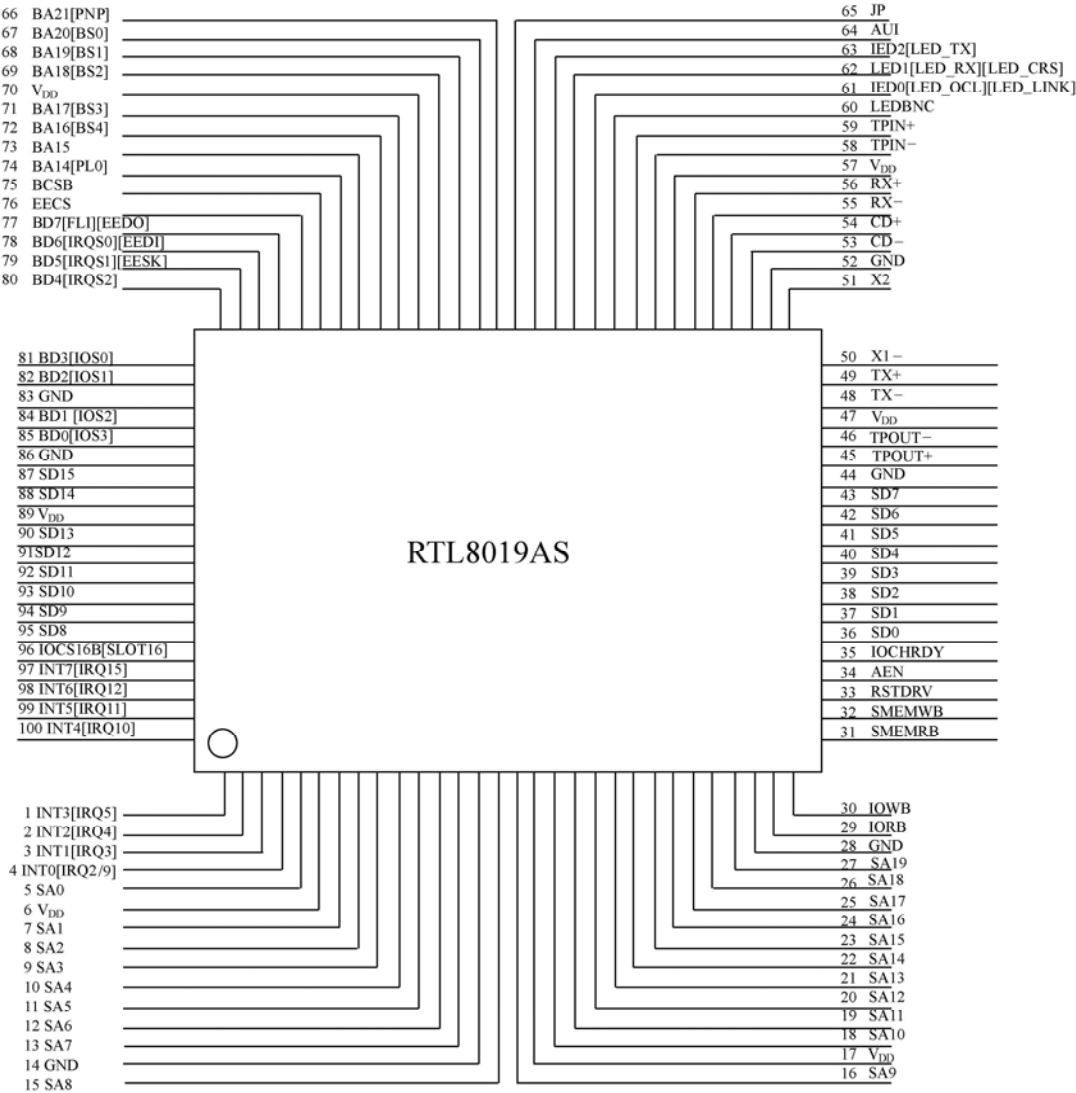


图 9-15 RTL8019AS 的引脚分布图

RTL8019AS 的引脚功能描述如下所示。

电源引脚如表 9-6 所示。

表 9-6 电源引脚功能说明

引 脚 号	引 脚 名 称	类 型	描 述
6, 17, 47, 57, 70, 89	VDD	P	+5 V
14, 28, 44, 52, 83, 86	GND	P	接地

ISA 总线接口引脚如表 9-7 所示。

表 9-7 ISA 总线接口引脚说明

引 脚 号	引 脚 名 称	类 型	描 述
34	AEN	I	地址使能，ISA 信号对有效的输入/输出信号必须是低电平
97~100, 1~4	INT0~INT7	O	中断请求总线。能够分别映射到 IRQ15、IRQ12、IRQ10、IRQ5、IRQ4、IRQ3 和 IRQ2/9。唯一一条线被选择在一个时间反映中断请求。RTL8019AS 也用这些引脚作为输入线，从而管理 ISA 总线上实际相应的中断线上的状态。结果记录在 INTR 寄存器中
35	IOCHRDY	O	低电平作为循环等待当前读/写指令
96	IOCS16B SLOT16	O	根据电源复位，以 IOCS16B 作为输入信号来检测 16 位或者 8 位是否在使用
29	IORB	I	输入/输出读/写指令端
30	IOWB	I	输入/输出读/写指令端
33	RSTDRV	I	ISA 总线上的硬件复位端，少于 800 ns 的高电平脉冲可以被忽略
5, 7~13, 15, 16, 18~27	SA0~SA19	I	地址总线。SA10 用来实现 PNP 端口的完全解码，地址为 279H 和 A79H
36~43, 87, 88, 90~95	SD0~SD15	I/O	数据总线
31	SMEMRB	I	存储器读命令
32	SMEMWB	I	存储器写命令

存储器接口引脚（包括 BROM 和 E²PROM）如表 9-8 所示。

表 9-8 存储器接口引脚说明

引 脚 号	引 脚 名 称	类型	描 述
75	BCSB	O	BROM 片选端。低电平有效,为读信号。当 SA19~SA14 和被选的 BROM 地址想匹配以及满足以下两个条件之一时，RTL8019AS 驱动其为低电平。 1. SMEMRB 为低电平； 2. SMEMRB 为低电平并且 RTL8019AS 的闪存读功能禁止
76	EECS	O	9346 片选。高电平有效，9346 读/写
66~69, 71~74,	BA21~BA14	O	BROM 地址
77~82, 84, 85	BD7~BA0	I/O	BROM 数据线
79	EESK	O	9346 串行数据时钟
78	EEDI	O	9346 串行数据输入
77	EEDO	I	9346 串行数据输出

下列引脚是为了定义跳跃者选项，如表 9-9 所示。它们在 RSTDRV 下降沿时被锁定，然后它们被用做 SRAM 总线，每个引脚被 100 kW 的内部电阻下拉。因此，当左开而且为高电平当其被 10 kΩ电阻上拉时，输入为低。

表 9-9 定义跳跃者选项

引 脚 号	引 脚 名 称	类 型	描 述
66	PNP	I	当 JP 为低电平时，RTL8019AS 被强制为即插即用模式 下列各项不需要注意 jumperless 模式（JP=低电平时）
72，71， 69～67	BS4～BS0	I	选择 BROM 大小和地址
85，84， 82～81	IOS3～IOS0	I	选择 I/O 地址
77，74	PL1～PL0	I	选择网络媒体类型
80～78	IRQS2～IRQS0	I	在 IN7～IN0 中选择一个中断
65	JP	I	当为高电平，将选择 jumper 模式；为低电平时，选择 jumperless 模式

媒体接口引脚如表 9-10 所示。

表 9-10 媒体接口引脚说明

引脚号	引 脚 名 称	类 型	描 述
64	AUI	I	用来检测在 AUI 接口的外部 MAU 的使用情况。输入对嵌入的 BNC 必须为低电平，对 MAU 必须为高电平。当输入为高电平，RTL8019AS 设置 AUI 位为 CONFIG0 并且驱动 LEDBNC 为低电平从而使 BNC 禁用；当该引脚未用时，应该接地
54，53	CD+，CD-	I	是从 MAU 来的微分输入信号的进位
56，55	RX+，RX-	I	是 AUI 接收端对 MAU 接收微分输入信号的进位
49，48	TPIN+， TPIN-	O	是一对进位微分的 tp 传输输出。输出曼彻斯特编码信号有预扭曲性，以防止在双绞线媒体的 overcharge，因此减少资源紧张
50	X1-	I	20 MHz 的晶体或者外部振荡器输入
51	X2+	O	晶体反馈输出。这个输出是位移的晶体连接方法。它必须是当 X1 在受外部振荡器驱动时左开的

LED 输出端口如表 9-11 所示。

表 9-11 输出端口

引 脚 号	引 脚 名 称	类 型	描 述
60	LEDBNC	O	当 RTL8019AS 媒体类型设置为 10BASE2 或者自动检测模式并且有链环测试失败时为高电平。用来控制对 CX MAU 的直流转换能量，而且连接到 LED 以表明所用媒体类型
61	LED0	O	当 LEDS0=0，作为 LED_COL； 当 LEDS0=1，作为 LED_LINK
62,63	LED1,LED2	O	当 LEDS1=0，这两个端作为 LED_RX 和 LED_TX； 当 LEDS=1，作为 LED_CRIS 和 MCSB

RTL8019AS 的第 65 引脚 JP 决定网卡使用哪种工作方式。RTL8019AS 有三种工作方式：第一种：跳线方式，网卡的 I/O 和中断由跳线决定；

第二种：即插即用方式，由软件进行自动配置 plug and play；
第三种：免跳线方式，网卡的 I/O 和中断由外接的 9346 里的内容决定。

当第 65 引脚 JP 为低电平时，RTL8019AS 工作在第二种或第三种方式，具体由 93C46 里的内容决定。通常用到的 RTL8019AS 网卡一般第 65 引脚为悬空的，引脚的输入状态为低电平，悬空的输入脚的电平为低电平，里面有一个 100 kΩ 的下拉电阻，网卡工作在第二，三种工作方式，需要使用 93C46 芯片。当第 65 引脚 JP 接高电平（接到 V_{CC} 或通过一个 10 kΩ 的上拉电阻）时，RTL8019AS 工作在跳线方式下，那么网卡的 I/O 和中断就不是用 93C46 的内容决定，这时不需要使用 93C46，芯片的 I/O 地址由 85、84、82、81（IOS3~IOS0）几个引脚决定。

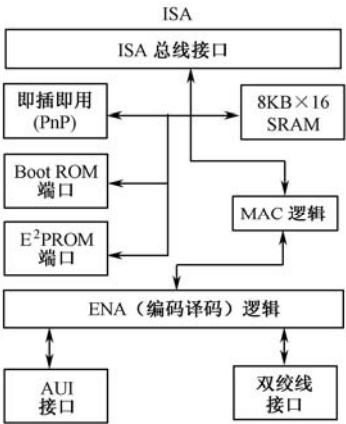


图 9-16 RTL8019AS 内部结构图

(2) 内部结构

RTL8019AS 内部可分为远程 DMA 接口、本地 DMA 接口、MAC（介质访问控制）逻辑、数据编码解码逻辑和其他端口，其内部结构如图 9-16 所示。

远程 DMA 接口是指单片机对 RTL8019AS 内部 RAM 进行读/写的总线，即 ISA 总线的接口部分。单片机收发数据只需对远程 DMA 操作。

本地 DMA 接口是 RTL8019AS 与网线的连接通道，完成控制器与网线的的数据交换。MAC（介质访问控制）逻辑完成以下功能：当单片机向网上发送数据时，先将一帧数据通过远程 DMA 通道送到 RTL8019AS 中的发送缓存区，然后发出传送命令；当

RTL8019AS 完成了上一帧的发送后，再开始此帧的发送，RTL819 接收到的数据经过 MAC 比较、CRC 校验后，由 FIFO 存到接收缓冲区；收满一帧后，以中断或寄存器标志的方式通知主处理器，FIFO 逻辑对收发数据进行 16 B 的缓冲，以减少对本地 DMA 请求的频率。

2. 内部RAM地址空间分配

RTL8019AS 内部有两块 RAM 区，一块 16 KB，地址为 0x4000~0x7fff；另一块 32B，地址为 0x0000~0x001f。RAM 按页存储，每 256 B 为一页。一般将 RAM 的前 12 页（0x4000~0x4bff）存储区作为发送缓冲区；后 52 页（0x4c00~0x7fff）存储区作为接收缓冲区。第 0 页叫做 Prom 页，只有 32 B，地址为 0x0000~0x001f，用于存储以太网物理地址。

要接收和发送数据包就必须通过 DMA 读/写 RTL8019AS 内部 16 KB 的 RAM，它本质上是双端口的 RAM，是指有两套总线连接到该 RAM，一套总线 RTL8019AS 读/写该 RAM，即本地 DMA；另一套总线是单片机读/写该 RAM，即远程 DMA。

3. I/O地址分配

RTL8019AS 具有 32 位输入/输出地址，地址偏移量为 00H~1FH，其中 00H~0FH 共 16 个地址，为寄存器地址。寄存器分为 4 页：PAGE0、PAGE1、PAGE2 和 PAGE3，由

RTL8019AS 的命令寄存器（command register, CR）中的 PS1、PS0 位来决定要访问的页。但与 NE2000 兼容的寄存器只有前 3 页，PAGE3 是由 RTL8019AS 自己定义的，对于其他兼容 NE2000 的芯片（如 DM9008）无效。远程 DMA 地址包括 10H~17H，都可以用做远程 DMA 端口，只要用其中的一个就可以了。复位端口包括 18H~1FH 共 8 个地址，用于 RTL8019AS 复位。

4. 寄存器描述

RTL8019AS 中的寄存器根据地址和功能可以分为两组，一组是与 NE2000 兼容的寄存器，另一组是即插即用寄存器。本例使用的是与 NE2000 兼容的寄存器。

与 NE2000 兼容的寄存器包括寄存器的 4 个页面，它们在 CR 中通过 PS0 和 PS1 被选择，每一页面包括 16 个寄存器，这些寄存器和 NE2000 兼容。RTL8019AS 为软件结构，以及为了增强特性还定义了一些寄存器，下面将介绍几组常用的寄存器。

（1）命令寄存器（CR）

CR 的地址为 00H，类型为可读/写，这个寄存器用来选择寄存器页面，能够使或者不能够使远程 DMA 操作和命令操作。主要内容如表 9-12 所示。

表 9-12 命令寄存器

PS1	PS0	RD2	RD1	RD0	TXP	STA	STP
MSB				LSB			

PS1、PS0：选择寄存器页。PS1PS0=00 时，选择第 0 页；PS1PS0=01 时，选择第 1 页；PS1PS0=10 时，选择第 2 页；PS1PS0=11 时，选择第 3 页。

RD2~RD0：表示要执行的功能。RD2~RD0=000 时，远程读内存；RD2~RD0=010 时，远程写内存；RD2~RD0=011 时，发送数据包；RD2 为 1 时，退出或者完成 DMA 操作。

TXP：置 1 表示发送数据包，发送完成或退出自动清零后复位。

STA、STP：用于启动或停止命令，STASTP=10 时，启动；STASTP=01 时，停止。

（2）中断状态寄存器（ISR）

ISR 的地址为第 0 页的 07H 时，类型为可读/写，这个寄存器反映网卡的状态。主机用来决定中断的原因。通过对相应的位写 1 来清除该位，它必须在上电源后清除。主要内容如表 9-13 所示。

表 9-13 中断状态寄存器

RST	RDC	CNT	OVW	TXE	RXE	PTX	PRX
MSB				LSB			

RST：当 NIC 输出设置状态时该位被设置；当启动对 CR 命令时该位被清除。当收到缓冲器溢出时该位被设置；当一个或者多个数据包从缓冲器中读取时该位被清除。

RDC：当远程 DMA 操作完成时被设置。

CNT：当一个或多个网络计数器的 MSB 设置完成时该位被设置。

OVW：当接收缓冲器用完时该位被设置。

TXE：由于过分冲突导致成组传送失败时对该位传输错误位设置。

RXE：当一个数据包接收有以下错误时该位被设置：

- CRC 错误；
- 帧同步错误；
- 丢失数据包。

PTX：该位表明无错误的数据包传送。

PRX：该位表明无错误的数据包接收。

(3) 中断屏蔽寄存器 (IMR)

IMR 的地址为第 0 页的 0FH 时，类型为只写；地址为第 2 页的 0FH 时，类型为只读。所有的位与 IST 寄存器各位相对应。POWER UP=0 时，设置某一位会使相应的中断打开。

(4) 数据结构寄存器 (DCR)

DCR 的地址为第 0 页的 0EH 时，类型为只写；地址为第 2 页的 0EH 时，类型为只读，具体内容如表 9-14 所示。

表 9-14 数据结构寄存器

—	FT1	FT0	ARM	LS	LAS	BOS	WTS
MSB				LSB			

位 7 的 “—” 表示取值始终为 1。

FT1、FT0：FIFO 片选位，为 1 或者 0。

ARM：自动远程初始化。ARM =0 时，发送数据包，命令不执行；ARM =1 时，发送数据包，命令执行；

LS：loopback 选择。LS =0 时，选择 loopback 模式，对 loopback 操作，TCR 为位 1 和位 2 必须编程；LS=1 时，正常操作。

LAS：该位必须设置为 0。NIC 仅仅支持双 16 位 DMA 模式。

BOS：字节顺序选择（不执行）。BOS=0 时，MS 字节放置在 MD15~MD8，LS 字节放置在 MD7~MD0（32XXX，80X86）；BOS =1 时，MS 字节放置在 MD7~MD0，LS 字节放置在 MD15~MD8（680X0）。

WTS：字传送选择。WTS =0 时，宽字节的 DMA 传送；WTS =1 时，宽字的 DMA 传送。

(5) 传输配置寄存器 (TCR)

TCR 的地址为第 0 页的 0DH 时，类型为只写；地址为第 2 页的 0DH 时，类型为只读，具体内容如表 9-15 所示。

表 9-15 TCR 传输配置寄存器

—	—	—	OFST	ATD	LB1	LB0	CRC
MSB				LSB			

位 7、位 6、位 5 的 “—” 表示始终是 1。

OFST：使能够冲突补偿。

ATD：使不能自动传输。ATD =0 时，正常操作；ATD =1 时，对 62 位不能正常工作的传送器多址混乱接收，对 63 位能正常工作的传送器多址混乱接收。

LB1、LB0：方式选择位，如表 9-16 所示。

表 9-16 方式选择位

LB1	LB0	方 式	备 注
0	0	0	正常操作
0	1	1	内部的 lookback
1	0	2	外部的 lookback
1	1	3	外部的 lookback

CRC：NIC CRC逻辑包含一个为了传送器CRC的发电机和一个为了接收器CRC的检验器。这个位控制CRC逻辑的活动。当此位被设置，CRC被传送器禁止，否则CRC 被传送器附加，如表9-17所示。

表 9-17 CRC

条 件		CRC 逻辑活动	
CRC 位	方式	CTC 发生器	CRC 检验器
0	正常	激活的	激活的
1	正常	不能	激活的
0	loopback	激活的	不能
1	loopback	不能	激活的

(6) 传输状态寄存器（TSR）

TSR 的地址为第 0 页的 04H 时，类型为只读。这个寄存器表明数据包传输的状态，具体内容如表 9-18 所示。

表 9-18 传输状态寄存器

OWC	CDH	—	CRS	ABT	COL	—	PTX
MSB				LSB			

OWC：在 51.2 μs 内被检测到冲突时该位被设置。数据传送和正常冲突一样重新传送。

CDH：在接着的传播中的接口频带开始的 6.4 μs 内 NIC 监测的冲突信号。当传送器传送信号失败时该位被设置。

CRS: 当传送器在传送过程中失去数据包时, 该丢失位被设置。

ABT: 它指出因为超额的冲突而导致 NIC 失败的传送。

COL: 它指出有其他系统在网络中时的传播冲突。

PTX: 该位表示无错误的传送。

(7) 接收结构寄存器 (RCR)

RCR 的地址为第 0 页的 0CH 时, 类型为只写; 地址为第 2 页的 0CH 时, 类型为只读, 具体内容如表 9-19 所示。

表 9-19 接收结构寄存器

—	—	MON	PRO	AJM	AB	AR	SEP
MSB				LSB			

位 7、位 6 的 “—” 始终是 1。

MON: 当监视器方式位被设置, 为了地址匹配将被检测, 如果 CRC 和帧同步不好, 数据包将被缓冲到存储器。

PRO: 当 PRO=1, 接收所有的有具体目标地址的数据包; 当 PRO=0, 具体的目标地址必须和在第 0~5 页中的程序节点地址匹配。

AJM: 当 AM=1, 含有多点传输目标地址的数据包被接收; 当 AM=0, 含有多点传输目标地址的数据包被拒绝。

AB: 当 AB=1, 含有广播目标地址的数据包被接收; 当 AB=0, 含有广播目标地址的数据包被拒绝。

AR: 当 AR=1, 长度小于 64 B 的数据包被接收; 当 AR=0, 长度小于 64 B 的数据包被拒绝。

SEP: 当 SEP=1, 含有收到错误信息的数据包被接收; 当 SEP=0, 含有收到错误信息的数据包被拒绝。

(8) 接收状态寄存器 (RSR)

RSR 的地址为第 0 页的 0CH 时, 类型为只读, 具体内容如表 9-20 所示。

表 9-20 接收状态寄存器

DFR	DIS	PHY	MPA	—	FAE	CRC	PRX
MSB				LSB			

DFR: 当传送器或者冲突被检测时被设置。

DIS: 接收器不能使用。当 NIC 输入监视方式, 该位被设置且接收器不能使用。监视方式结束后接收器使能时该位被重置。

PHY: 当接收数据包含有多点传输目标或者广播目标地址时, PHY 位被设置; 当接收数据包含有具体目标地址时, 该位被重置。

MPA: 当下一个数据包因为缺少接收缓冲器或者 NIC 为监视方式, 而不能被接收时, 该丢失数据包位被设置, 增加 CNTR2 计数器。

FAE: 帧同步错误位。反映下一个数据包在字节边界未结束, 以及 CRC 和最后一个字节位不匹配, 增加 CNTR0 计数器。

CRC: CRC 错误位, 反映接收到的数据包含有 CRC 错误。当有 FAE 错误时该位也设置, 增加 CNTR1 计数器。

PRX: 此位表明接收到的数据包无错误。

(9) 其他寄存器

CLDA0、CLDA 1: 当前局部 DMA 寄存器。地址为第 0 页的 01H 或 02H 时, 类型为只读。通过读这两个寄存器可以得到当前 DMA 地址。

PSTART: 页面开始寄存器。地址为第 0 页的 01H 时, 类型为只写; 地址为第 2 页的 01H 时, 类型为只读。该寄存器用来设置接收缓冲器的开始页面地址。

PSTOP: 页面停止寄存器。地址为第 0 页的 02H 时, 类型为只写; 地址为第 2 页的 02H 时, 类型为只读。该寄存器用以设置接收缓冲器停止页面寄存器地址。在 8 位方式下 PSTOP 寄存器不应该超过 0x60, 在 6 位方式下 PSTOP 寄存器应该不超过 0x80。

BNRY: 边界寄存器。地址为第 0 页的 03H 时, 类型为只读/写, 该寄存器是用来放置接收缓冲器的重写。它代表性的作用是作为接收缓冲器最后页面的指针。

TPSR: 传输页面开始寄存器。地址为第 0 页的 04H 时, 类型为只写, 该寄存器用来设置传输数据包开始页面地址。

TBCR0、TBCR 1: 传输字节计算寄存器。地址为第 0 页的 05H 和 06H 时, 类型为只写, 该寄存器用来设置传输数据包的字节计数。

NCR: 冲突数寄存器。地址为第 0 页的 05H 时, 类型为只读, 该寄存器用来记录在数据包传输过程中的冲突节点数。

FIFO: 先进先出寄存器。地址为第 0 页的 06H 时, 类型为只读, 该寄存器允许主机检查在 loopback 后的 FIFO 内容。

CRDA0、CRDA 1: 当前远程 DMA 寄存器。地址为第 0 页的 08 和 09H 时, 类型为只读。这两个寄存器包括当前远程 DMA 地址。

RSAR0、RSAR01: 远程起始地址寄存器。地址为第 0 页的 08 和 09H 时, 类型为只写。这两个寄存器用来设置远程 DMA 起始地址。

RBCR0、RBCR: 远程字节数寄存器。地址为第 0 页的 0AH 和 0BH 时, 类型为只写。这两个寄存器用来设置远程 DMA 数据字节数。

CNTR0: 帧同步错误计数寄存器。地址为第 0 页的 0DH 时, 类型为只读。

CNTR1: CRC 错误数记录寄存器。地址为第 0 页的 0EH 时, 类型为只读。

CNTR2: 丢失数据包数记录寄存器。地址为第 0 页的 0FH 时, 类型为只读。

PAR0~5: 实际地址寄存器。地址为第 1 页的 01H~06H 时, 类型为读/写。这些寄存器包括以太网节点地址, 用来对目标地址数据包进行比较来确定接收或者拒绝接收。

CURR: 当前页面寄存器。地址为第 1 页的 07H 时, 类型为读/写。该寄存器指出首先接收缓冲器页面地址, 这个页面用来接收数据包。

MAR0~MAR7: 多点传输地址寄存器。地址为第 1 页的 08H~0FH 时, 类型为读/写。这些寄存器提供被 CRC 逻辑变位无用的多点地址的滤波器。

9.4.5 硬件电路设计

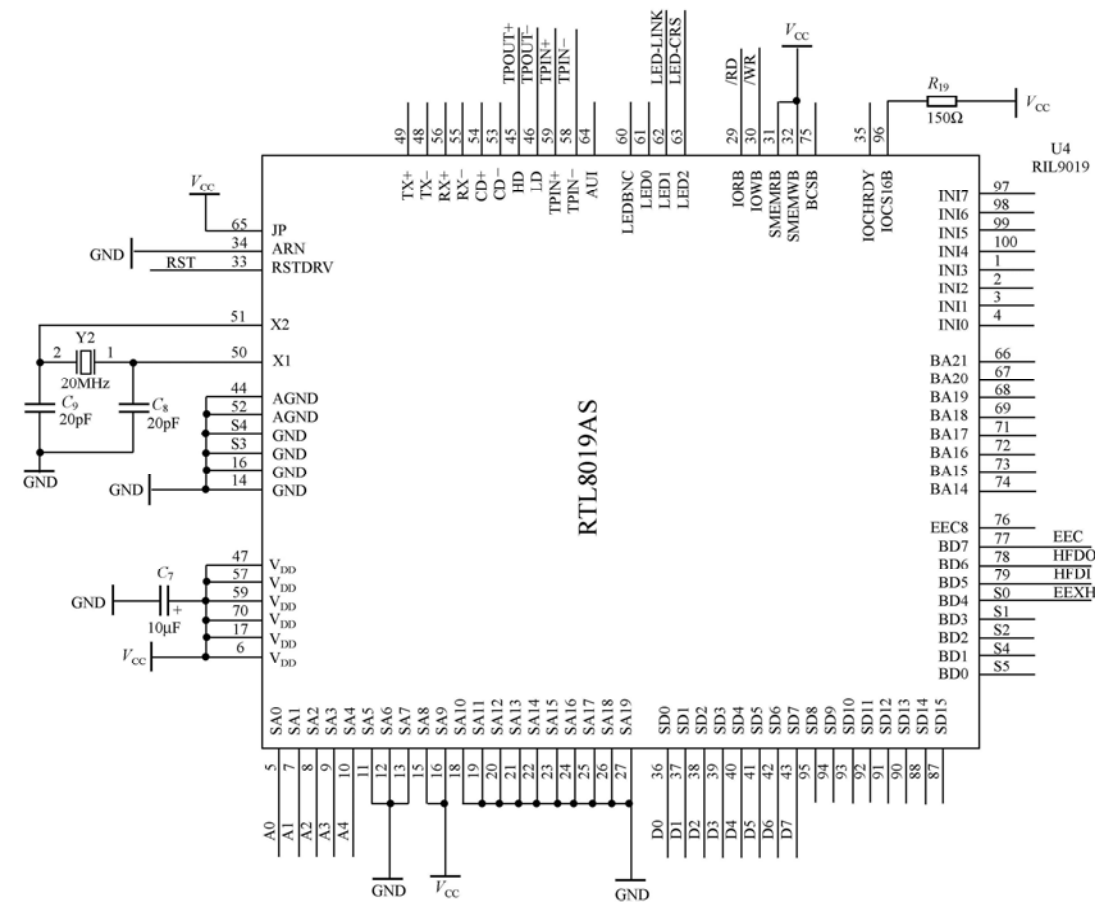
C51 单片机控制 RTL8019AS 实现以太网通信的接口的电路，如图 9-17 所示。

接口电路中用到的主要芯片有 AT89C52、RTL8019AS、93C46（64×16 位的 E²PROM）、74HC573（8 位锁存）、HM62256（32 KB 的 RAM）。为分配好地址空间，采用对 93C46 进行读（或写）操作来设置 RTL8019AS 的端口 I/O 基地址和以太网物理地址。

RTL8019AS 的地址为 20 位，RTL8019AS 的地址空间为 00300H~0031FH，用二进制表示 00300H~0031FH，可以发现第 19 位到第 5 位是固定的：000000000011000。RTL8019AS 的 20 根地址线的连接方式 SA0~SA19 如表 9-21 所示。

表 9-21 RTL8019AS 地址线的连接方式

引脚名称	连接方式
SA19~SA10	接地
SA9~SA8	接单片机 P2 口的 P2.7，即地址总线 ADDR15
SA7~SA5	接地
SA4~SA0	对应为地址总线的 ADDR0~ADDR4



(a) 以太网控制芯片 RTL8019AS 部分原理图

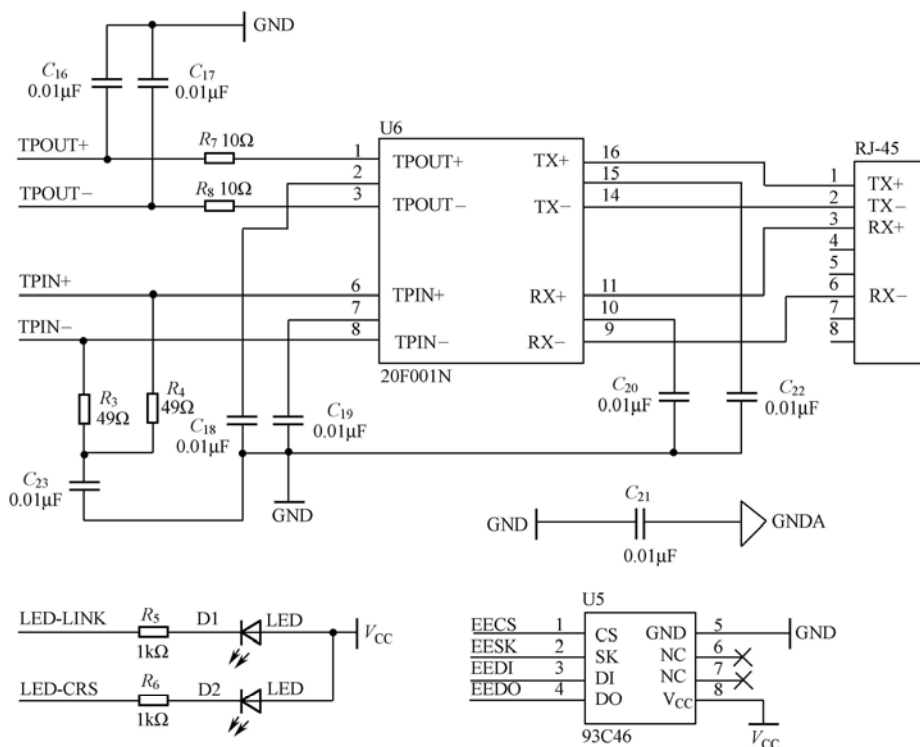
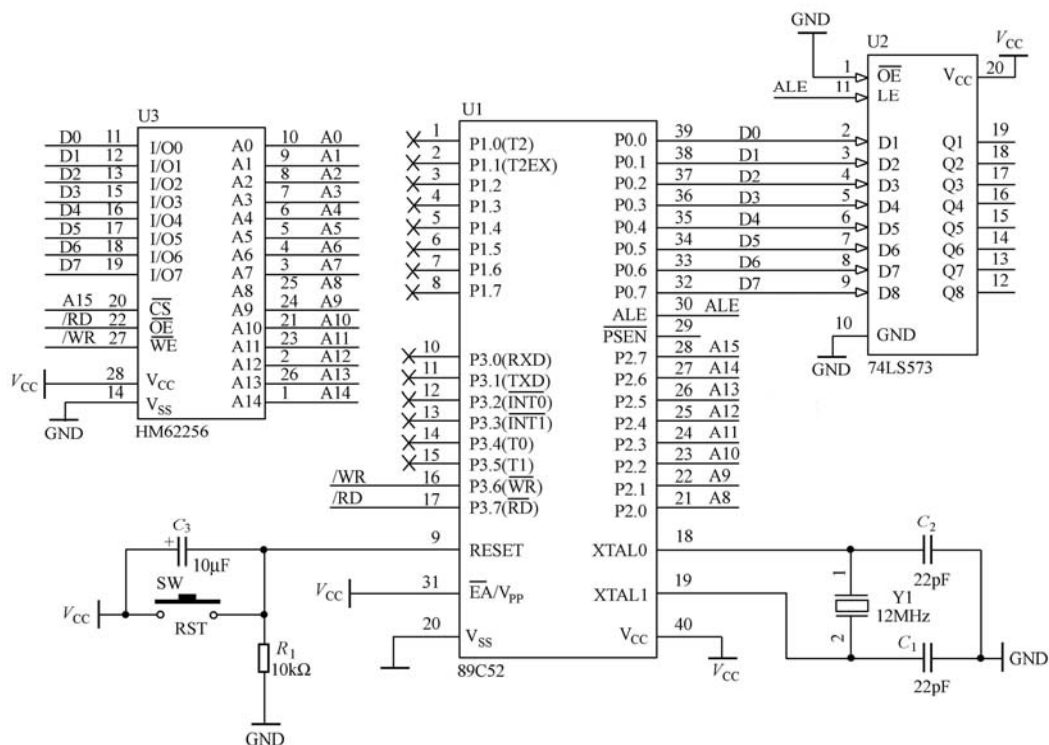


图 9-17 以太网通信接口的电路图

93C46 是采用 4 线 SPI 串行接口的 Serial E²PROM，容量为 1 KB（64×16 位），主要用来保存 RTL8019AS 的配置信息。00H~03H 的地址空间用于存储 RTL8019AS 内配置寄存器 CONFIG1~4 的上电初始化值；地址 04H~11H 存储网络节点地址，即物理地址；地址 12H~7FH 内存储即插即用的配置信息。RTL8019AS 通过引脚 EECS、EESK、EEDI 控制 93C46 的 CS、SK、DI 引脚，通过 EEDO 接收 93C46 的 DO 引脚的状态。RTL8019AS 复位后读取 93C46 的内容并设置内部寄存器的值，如果 93C46 中内容不正确，则 RTL8019AS 无法正常工作。先通过编程器，如 ALL07 把配置好的数据写入 93C46，再焊入电路。

9.4.6 软件设计

本例的程序代码只给出了对 RTL8019AS 进行初始化部分。

1. RTL8019AS的初始化

对 RTL8019AS 进行初始化的具体步骤如下。

初始化第 0 页与第 1 页的相关寄存器，因为 RTL8019AS 第 2 页的寄存器是只读的，所以不能进行设置，而第 3 页的寄存器与 NE2000 是不兼容的，所以也不用设置。

- ① CR=0x21，选择第 0 页的寄存器；
- ② TPSR=0x45，发送页的起始页地址，初始化为指向第一个发送缓冲区的页，即 0x40；
- ③ PSTART=0x4c，PSTOP=0x80，构造缓冲环：0x4c~0x80；
- ④ BNRY=0x4c，设置指针；
- ⑤ RCR=0xcc，设置接收配置寄存器，使用接收缓冲区，仅接收自己地址的数据包（以及广播地址数据包）和多点播送地址包，小于 64 B 的包丢弃，校验错的数据包不接收；
- ⑥ TCR=0xe0，设置发送配置寄存器，启用 CRC 自动生成和自动校验，工作在正常模式；
- ⑦ DCR=0xc8，设置数据配置寄存器，使用 FIFO 缓存，普通模式，8 位数据 DMA；
- ⑧ IMR=0x00，设置中断屏蔽寄存器，屏蔽所有的中断；
- ⑨ CR=0x61，选择第 1 页的寄存器；
- ⑩ CURR=0x4d，CURR 是 RTL8019AS 写内存的指针，指向当前正在写的页的下一页，初始化时指和 0x4c+1=0x4d；
- ⑪ 设置多址寄存器 MAR0~MAR5，均设置为 0x00；
- ⑫ 设置网卡地址寄存器 PAR0~PAR5；
- ⑬ CR=0x22，选择第 1 页的寄存器，进入正常工作状态。

2. 程序说明

单片机控制 RTL8019AS 以太网控制芯片，实现以太网接口电路的程序代码如下。

```
#include <reg52.h>           // 引用标准库的头文件
#include <absacc.h>
#include <stdio.h>
```

```

#define uchar unsigned char
#define uint unsigned int

#define REG00 XBYTE[0x8000]           // 端口 300H, 命令寄存器 CR
#define REG01 XBYTE[0x8001]           // 端口 301H
#define REG02 XBYTE[0x8002]           // 端口 302H
#define REG03 XBYTE[0x8003]           // 端口 303H
#define REG04 XBYTE[0x8004]           // 端口 304H
...
#define REG1e XBYTE[0x801e]           // 端口 31EH
#define REG1f XBYTE[0x801f]           // 端口 31FH

void delay(uint r);
void 8019Rst();
void Select_Page(uchar pagenum);
void ClearISR();
void GetPhyAddress();
void 8019Init();

void main( )
{
    delay(1000);                       // 延时 1 s
    8019Rst();                          // RTL8019AS 复位
    ClearISR();                         // 清除 ISR 寄存器
    8019Init();                         // RTL8019AS 初始化
    while(1)
    {
    }
}

//延时函数
void delay(uint r)
{
    uint a;
    while(r--)
    {
        for (a=0;a<125;a++)
        {}
    }
}

```

//RTL8019AS 热复位函数

void 8019Rst()

```
{
    uchar i,tmp;
    tmp = REG1f;           //读复位端口
    REG1f = tmp;           //写复位端口
    for(i=0;i<250;i++);    //延时
}
```

//通过 CR 寄存器的 PS1 和 PS0 设置寄存器页

void Select_Page(uchar pagenum)

```
{
    uchar tmp;
    tmp = REG00;
    tmp = tmp&0x3B;        //TXP 位在不发送数据包时要置 0
    pagenum = pagenum<<6;
    tmp = tmp|pagenum;
    REG00 = tmp;
}
```

//初始化 RTL8019AS 函数

void 8019Init()

```
{
    REG00 = 0x21; // 选择第 0 页的寄存器，网卡停止运行
    REG01 = 0x4c; // 寄存器 PSTART，设置接收缓冲区的起始页地址
    REG02 = 0x80; // 寄存器 PSTOP，设置接收缓冲区的结束页地址
    REG03 = 0x4c; // 寄存器 BNRY，设置为指向第一个接收缓冲区的页
    REG04 = 0x40; // 寄存器 TPSR，发送起始页地址初始化为指向第一个发送缓冲区的页
    REG0c = 0xcc;
    REG0d = 0xe0; // 发送配置寄存器 TCR，设置为启用 CRC 自动生成和校验，正常模式工作
    REG0e = 0xc8;
    REG0f = 0x00; // 中断屏蔽寄存器 IMR，设置为屏蔽所有中断
    Select_Page(1); // 选择第 1 页的寄存器
    REG07=0x4d; // 寄存器 CURR，设置为指向当前正在写的页的下一页

    REG08 = 0x00; // 多址地址寄存器 MAR0
    REG09 = 0x00; // 多址地址寄存器 MAR1
    REG0a = 0x00; // 多址地址寄存器 MAR2
    REG0b = 0x80; // 多址地址寄存器 MAR3
    REG0c = 0x00; // 多址地址寄存器 MAR4
    REG0d = 0x00; // 多址地址寄存器 MAR5
}
```

```

REG0e = 0x00; // 多址地址寄存器 MAR6
REG0f = 0x00; // 多址地址寄存器 MAR7

GetPhyAddress(); // 获取以太网物理地址

REG00 = 0x22; // 选择第 0 页寄存器，执行命令
}

//上电后清除 ISR 寄存器函数
void ClearISR()
{
    Select_Page(0);
    REG07 = REG07|0xff;
}

//获取以太网物理地址函数
void GetPhyAddress()
{
    uchar tmp;

    Select_Page(0); // 选择第 0 页
    REG08 = 0; // 远程 DMA 起始地址低位寄存器 RSAR0
    REG09 = 0; // 远程 DMA 起始地址高位寄存器 RSAR1
    REG0a = 12; // 远程 DMA 计数器低位寄存器 RBCR0
    REG0b = 0; // 远程 DMA 计数器高位寄存器 RBCR1

    REG00 = 0x0a; // 远程 DMA，启动命令

    Select_Page(1); // 选择第 1 页
    tmp = REG10; // 读一个字节
    REG01 = tmp; // 写入 PAR0
    tmp = REG10; // 读取一个重复的字节，这个字节被丢弃
    tmp = REG10; // 读取一个字节
    REG02 = tmp; // 写入 PAR1
    tmp = REG10; // 读取一个重复的字节，这个字节被丢弃
    tmp = REG10; // 读取一个字节
    REG03 = tmp; // 写入 PAR2
    tmp = REG10; // 读取一个重复的字节，这个字节被丢弃
    tmp = REG10; // 读取一个字节
    REG04 = tmp; // 写入 PAR3
    tmp = REG10; // 读取一个重复的字节，这个字节被丢弃

```

```

tmp = REG10;           // 读取一个字
REG05 = tmp;           // 写入 PAR4
tmp = REG10;           // 读取一个重复的字节，这个字节被丢弃
tmp = REG10;           // 读取一个字
REG06 = tmp;           // 写入 PAR5
}

```

9.5 低频信号发生器输出

信号发生器又称做信号源或振荡器，在生产实践和科技领域中有着非常广泛的应用。可以用三角函数方程式来表示各种波形曲线。但信号发生器 **OUTPNT** 只是需要的频率,没有经过后级放大，即输出阻抗比较大，而低频信号发生器有后级放大，输出阻抗比较小,可以直接带负载。在工业、农业、生物医学等领域内，如高频感应加热、熔炼、淬火、超声诊断、核磁共振成像等，都需要功率或大或小、频率或高或低的振荡器。

9.5.1 实例说明

本例所设计的低频信号发生器要求能输出频率范围在 0.1~50Hz 的正弦波、三角波和方波信号，其中正弦波和三角波信号可以用按键选择输出。

9.5.2 DAC0832 介绍

DAC0832 是 CMOS 工艺制造的 8 位数/模（D/A）转换器，属于 8 位电流输出型 D/A 转换器，转换时间为 1 μs，片内带输入数字锁存器，与单片机完全兼容，具有价格低廉，接口简单，转换控制容易等优点，在单片机应用系统中得到了广泛的应用。

1. DAC0832 引脚说明

DAC0832 的引脚如图 9-18 所示，其引脚说明如表 9-22 所示。

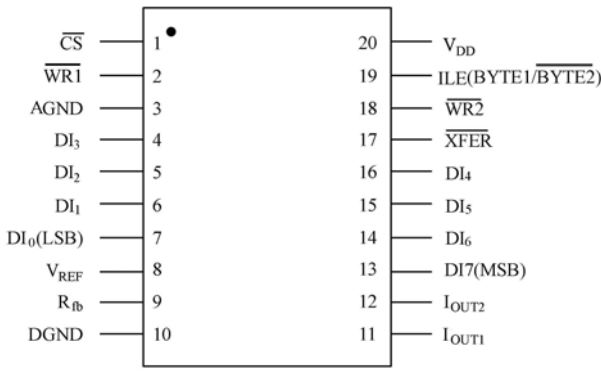


图 9-18 DAC0832 的引脚图

表 9-22 DAC0832 引脚说明

引 脚 名 称	说 明
DI ₀ ~DI ₇	数据输入，TLL 电平
ILE	数据锁存允许控制信号输入，高电平有效
$\overline{\text{CS}}$	片选信号输入线，低电平有效
$\overline{\text{WR1}}$	为输入寄存器的写选通信号
$\overline{\text{XFER}}$	数据传输控制信号输入，低电平有效
$\overline{\text{WR2}}$	为 DAC 寄存器写选通输入
I _{OUT1}	电流输出。当输入全为 1 时，I _{OUT1} 最大
I _{OUT2}	电流输出。其值与 I _{OUT1} 之和为一个常数
R _{fb}	反馈信号输入，芯片内部有反馈电阻
V _{DD}	电源输入（+5~+15 V）
V _{REF}	基准电压输入（-10~+10V）
AGND	模拟地，模拟信号和基准电源的参考地
DGND	数字地，两种地线在基准电源处共地比较好

2. DAC0832 的时序及内部结构图

DAC0832 的时序如图 9-19 所示。

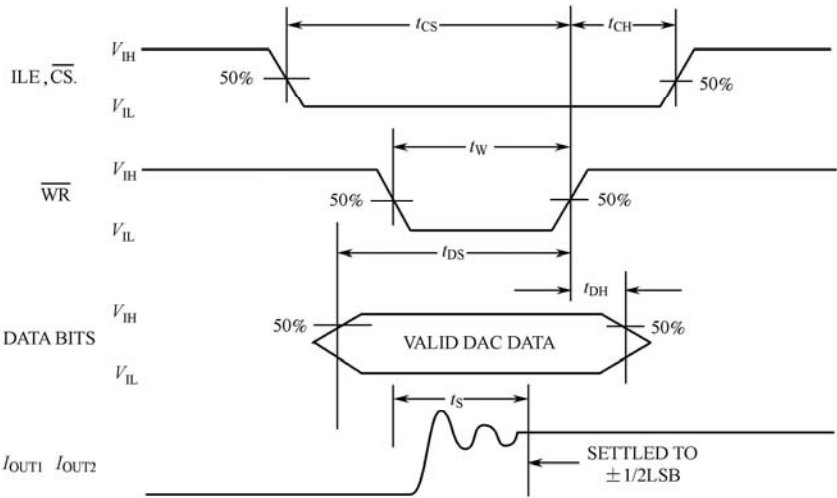


图 9-19 DAC0832 的时序图

DAC0832 的内部结构如图 9-20 所示。

注意：当使能端 $\overline{\text{LE}}=1$ 的时候，输出 Q 就等于输入 D；当使能端 $\overline{\text{LE}}=0$ 的时候，输出 Q 就会被锁存。

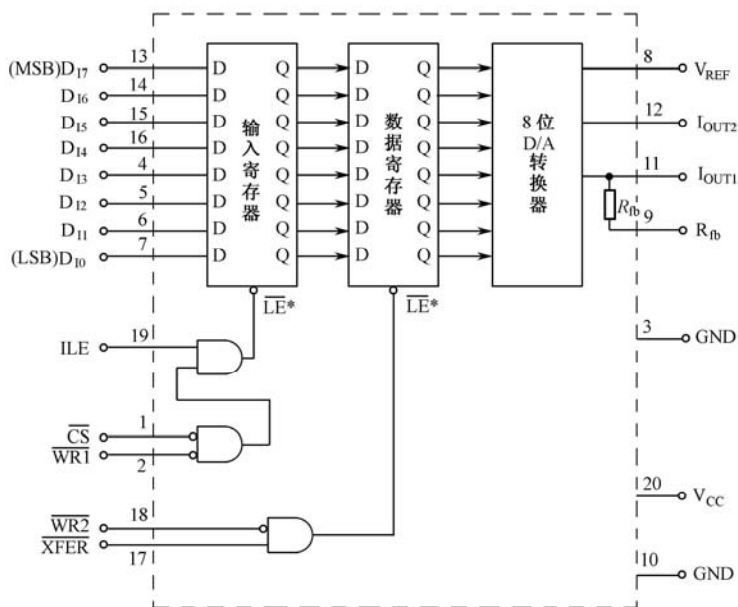


图 9-20 DAC0832 的内部结构框图

3. DAC0832 的几种常用典型应用

图 9-21、图 9-22、图 9-23、图 9-24、图 9-25 分别是 DAC0832 的 5 种常用典型应用的原理图。

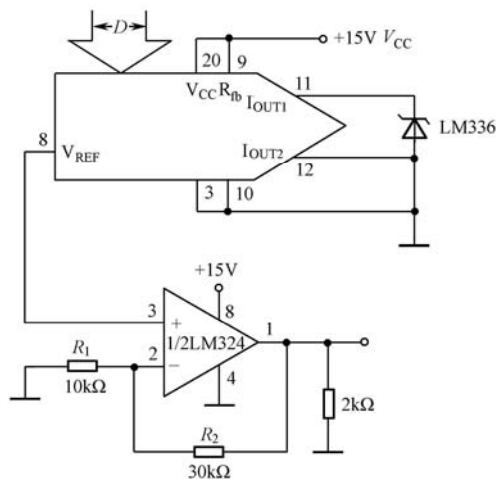


图 9-21 DAC0832 典型应用 (1)

在图 9-21 中

$$V_{\text{REF}} = 2.5V_{\text{CC}}$$

$$V_{\text{OUT}} = +2.5V_{\text{CC}} \left(1 + \frac{R_2}{R_1}\right) \left(\frac{D}{256}\right)$$

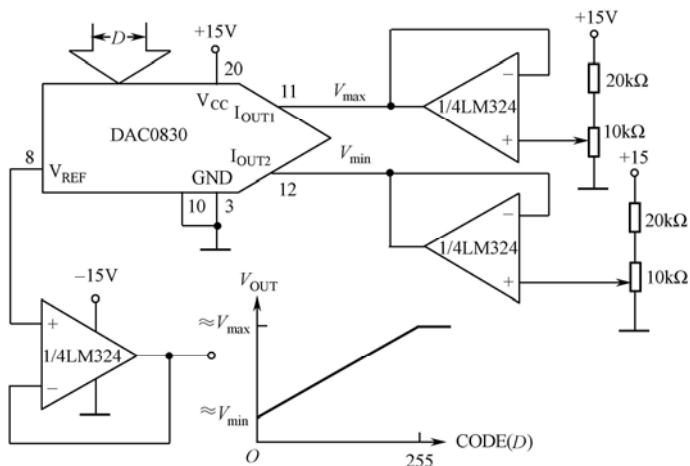


图 9-22 DAC0832 典型应用 (2)

在图 9-22 中

- 仅需要一个 15 V 的电压输入；
- 不需要调零；
- 输出范围在 0~+5 V 之间。

$$V_{OUT} = \frac{D}{256}(V_{max} - V_{min}) + \frac{255}{256}V_{min}$$

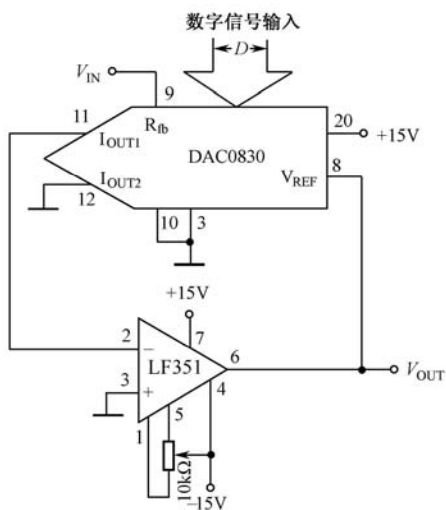


图 9-23 DAC0832 典型应用 (3)

在图 9-23 中

$$V_{OUT} = \frac{-V_{IN}(256)}{D}$$

随着输入从全 1 到全零的递减变化，从输入到输出的反馈电阻从 15 kΩ 到无穷大之间变化。

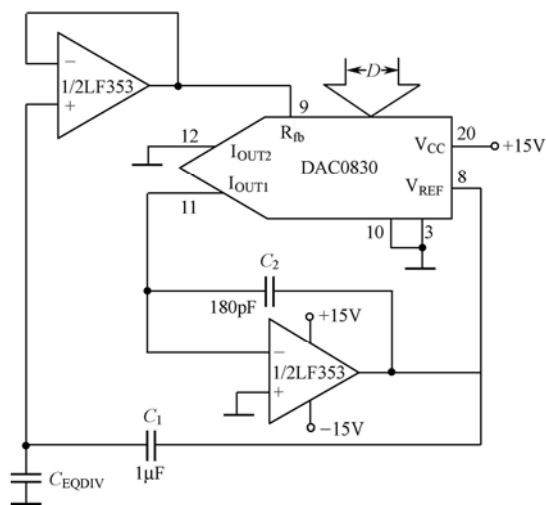


图 9-24 DAC0832 典型应用 (4)

图 9-24 为一个简易的电容倍增器，其中

$$C_{\text{EQUIV}} = C_1 \left(1 + \frac{256}{D} \right)$$

通过电容 C_{EQUIV} 的最大电压的极限是 $\frac{V_{\text{Omax}}(\text{opamp})}{1 + \frac{256}{D}}$ 。 C_2 的作用是提高运算放大器的稳

定时间。

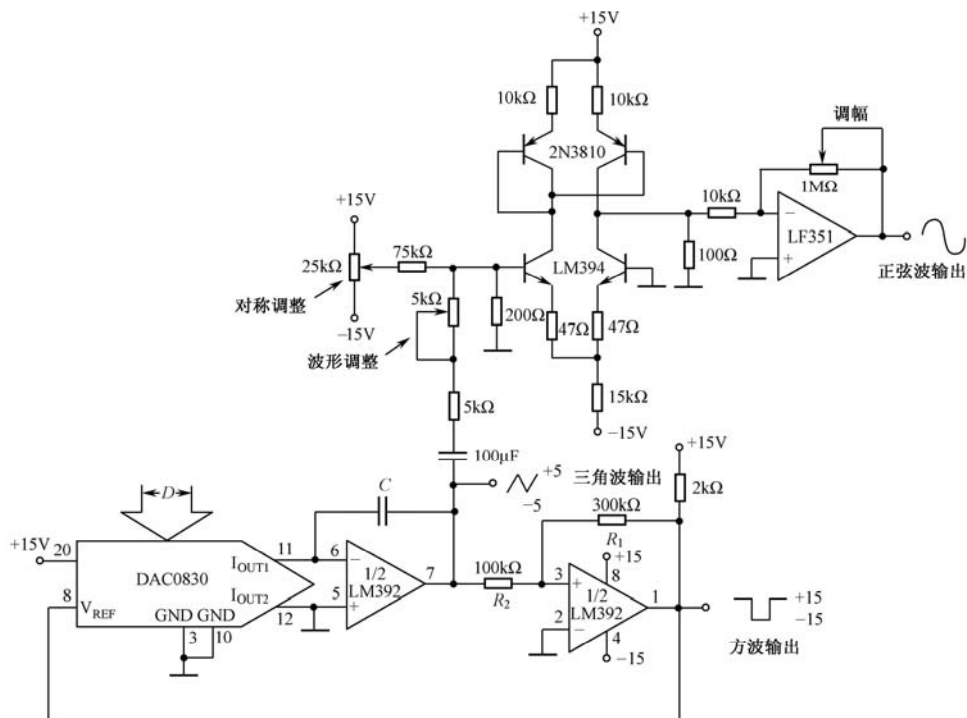


图 9-25 DAC0832 典型应用 (5)

图 9-25 是利用数/模转换器实现的信号发生器。其中，数模转换器控制正弦波、方波和三角波的输出频率。

4. DAC0832 的主要性能

- ① 输入数据为 8 位;
- ② 采用 CMOS 工艺, 所有引脚的逻辑电平与 TTL 兼容;
- ③ 数据输入可以采用双缓冲, 单缓冲或直通方式;
- ④ 转换时间 $1\ \mu\text{s}$;
- ⑤ 精度 $\pm 1\ \text{LSB}$;
- ⑥ 分辨率: 8 位;
- ⑦ 单一电源: $5\sim 15\ \text{V}$, 功耗 $20\ \text{mW}$;
- ⑧ 参考电压: $+10\sim -10\ \text{V}$ 。

硬件设计分为控制部分和数/模转换部分。

控制芯片使用 C51 单片机。控制系统按最小化工作模式设计, P3.0~P3.2 口接 3 个按键, 其中 T1 为频率增加键, T2 为频率减小键, T3 为正弦波与三角波选择按键。P1 口输出波形数据, 其中 P1.0 与 P1.1 需外接上拉电阻。控制部分硬件原理如图 9-26 所示。

图 9-26 低频信号发生器的单片机控制部分电路图

2. 数/模转换部分

DAC0832 与单片机结成数据直接写入方式，当单片机把一个数据直接写入 DAC 寄存器时，DAC0832 的输出模拟电压信号随之变化。利用 D/A 转换器可以产生各种波形，如方波，三角波，锯齿波等以及它们组合产生的复合波形和不规则波形。这些复合波形利用标准的测试设备是很难产生的。数/模转换部分硬件电路图如图 9-27 所示。

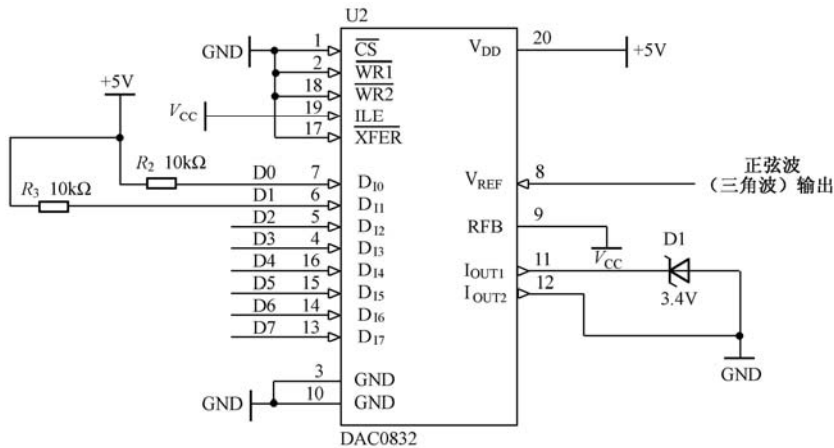


图 9-27 低频信号发生器的数/模转换部分电路图

9.5.4 软件设计

软件设计分为系统初始化子函数、键盘扫描子函数、波形数据产生子函数和主函数 4 个部分。

1. 初始化子函数

初始化子函数的主要工作是设置定时器的的工作模式，初值预置，开中断和打开定时器。定时器 T1 工作与 16 定时模式，单片机按定时时间重复地把波形数据送到 DAC0832 的寄存器。程序流程如图 9-28 所示。



图 9-28 初始化子函数程序流程图

2. 键盘扫描子函数

检查 3 个按键中是否有键按下，若有键按下，则执行相应的功能。这 3 个按键分别用于频率增加、频率减小和正弦波与三角波的选择功能。流程如图 9-29 所示。

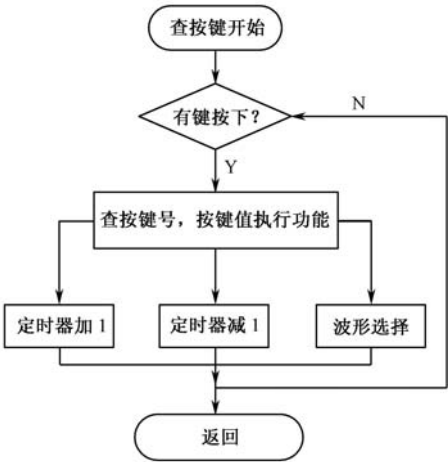


图 9-29 键扫描子函数程序流程图

3. 波形数据产生子函数

波形数据产生函数是定时器 T1 的中断程序。当定时器计数溢出时，发生一次中断。当发生中断时，单片机按次序将波形数据表的波形数据一一送入 DAC0832，DAC0832 根据输入的数据大小输出对应的电压。波形数据产生子函数流程如图 9-30 所示。

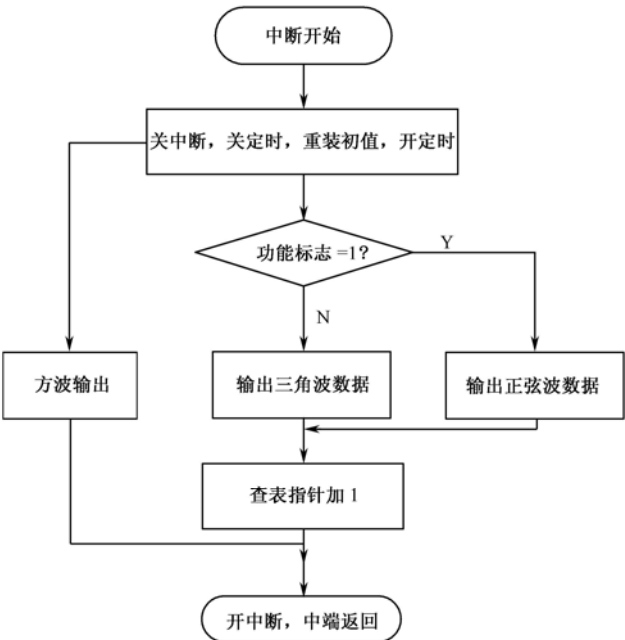


图 9-30 波形数据产生子函数流程图

4. 主函数

主函数用来进行上电初始化，并在程序运行中不断查询按键情况，并执行相应的功能。

5. 程序代码

程序代码如下：

```
//低频信号源
//2 MHz 晶振， AT89C2051
#include<reg51.h>
#define uchar unsigned char
#define uint unsigned int

#define key P3 //键盘口
#define output P1 //正弦波等数据输出口
//
uint data Timer1 = 65535; //T1 的 16 位定时器初值
uchar data keyword, n = 0; //键值存放，查表指针
bdata sinthr;
sbit sin_thr = sinthr^0; //正弦波/三角波标志（1 为正弦）
sbit ww = key^7; //方波输出口
//初始化函数
start()
{
    key = 0xff;
    output = 0x00;
    Timer1 = 65535; //按键输入状态，波形输出为 0
    TH1 = Timer1/256; //定时初值（T1 用）
    TL1 = Timer1%256;
    TMOD = 0X11;
    ET1 = 1;
    TR1 = 1; //16 位定时模式，T1 中断允许，开定时器，
    EA = 1; //开总中断
}
//三角波数据表
uchar code thr_table[256]=
    { 0x80,0x81,0x82,0x83,0x84,0x85,0x86,0x87,
      0x88,0x89,0x8A,0x8B,0x8C,0x8D,0x8E,0x8F,
      0x90,0x91,0x92,0x93,0x94,0x95,0x96,0x97,
      0x98,0x99,0x9A,0x9B,0x9C,0x9D,0x9E,0x9F,
      0xA0,0xA1,0xA2,0xA3,0xA4,0xA5,0xA6,0xA7,
```

```

0xA8,0xA9,0xAA,0xAB,0xAC,0xAD,0xAE,0xAF,
0xB0,0xB1,0xB2,0xB3,0xB4,0xB5,0xB6,0xB7,
0xB8,0xB9,0xBA,0xBB,0xBC,0xBD,0xBE,0xBF,
0xBF,0xBE,0xBD,0xBC,0xBB,0xBA,0xB9,0xB8,
0xB7,0xB6,0xB5,0xB4,0xB3,0xB2,0xB1,0xB0,
0xAF,0xAE,0xAD,0xAC,0xAB,0xAA,0xA9,0xA8,
0xA7,0xA6,0xA5,0xA4,0xA3,0xA2,0xA1,0xA0,
0x9F,0x9E,0x9D,0x9C,0x9B,0x9A,0x99,0x98,
0x97,0x96,0x95,0x94,0x93,0x92,0x91,0x90,
0x8F,0x8E,0x8D,0x8C,0x8B,0x8A,0x89,0x88,
0x87,0x86,0x85,0x84,0x83,0x82,0x81,0x80,
0x7F,0x7E,0x7D,0x7C,0x7B,0x7A,0x79,0x78,
0x77,0x76,0x75,0x74,0x73,0x72,0x71,0x70,
0x6F,0x6E,0x6D,0x6C,0x6B,0x6A,0x69,0x68,
0x67,0x66,0x65,0x64,0x63,0x62,0x61,0x60,
0x5F,0x5E,0x5D,0x5C,0x5B,0x5A,0x59,0x58,
0x57,0x56,0x55,0x54,0x53,0x52,0x51,0x50,
0x4F,0x4E,0x4D,0x4C,0x4B,0x4A,0x49,0x48,
0x47,0x46,0x45,0x44,0x43,0x42,0x41,0x40,
0x40,0x41,0x42,0x43,0x44,0x45,0x46,0x47,
0x48,0x49,0x4A,0x4B,0x4C,0x4D,0x4E,0x4F,
0x50,0x51,0x52,0x53,0x54,0x55,0x56,0x57,
0x58,0x59,0x5A,0x5B,0x5C,0x5D,0x5E,0x5F,
0x60,0x61,0x62,0x63,0x64,0x65,0x66,0x67,
0x68,0x69,0x6A,0x6B,0x6C,0x6D,0x6E,0x6F,
0x70,0x71,0x72,0x73,0x74,0x75,0x76,0x77,
0x78,0x79,0x7A,0x7B,0x7C,0x7D,0x7E,0x7F};

```

//正弦波数据表

```
uchar code sin_table[256]=
```

```

{ 0x80,0x83,0x85,0x88,0x8A,0x8D,0x8F,0x92,
  0x94,0x97,0x99,0x9B,0x9E,0xA0,0xA3,0xA5,
  0xA7,0xAA,0xAC,0xAE,0xB1,0xB3,0xB5,0xB7,
  0xB9,0xBB,0xBD,0xBF,0xC1,0xC3,0xC5,0xC7,
  0xC9,0xCB,0xCC,0xCE,0xD0,0xD1,0xD3,0xD4,
  0xD6,0xD7,0xD8,0xDA,0xDB,0xDC,0xDD,0xDE,
  0xDF,0xE0,0xE1,0xE2,0xE3,0xE3,0xE4,0xE4,
  0xE5,0xE5,0xE6,0xE6,0xE7,0xE7,0xE7,0xE7,
  0xE7,0xE7,0xE7,0xE7,0xE6,0xE6,0xE5,0xE5,
  0xE4,0xE4,0xE3,0xE3,0xE2,0xE1,0xE0,0xDF,
  0xDE,0xDD,0xDC,0xDB,0xDA,0xD8,0xD7,0xD6,
  0xD4,0xD3,0xD1,0xD0,0xCE,0xCC,0xCB,0xC9,

```



```

0xC7,0xC5,0xC3,0xC1,0xBF,0xBD,0xBB,0xB9,
0xB7,0xB5,0xB3,0xB1,0xAE,0xAC,0xAA,0xA7,
0xA5,0xA3,0xA0,0x9E,0x9B,0x99,0x97,0x94,
0x92,0x8F,0x8D,0x8A,0x88,0x85,0x83,0x80,
0x7D,0x7B,0x78,0x76,0x73,0x71,0x6E,0x6C,
0x69,0x67,0x65,0x62,0x60,0x5D,0x5B,0x59,
0x56,0x54,0x52,0x4F,0x4D,0x4B,0x49,0x47,
0x45,0x43,0x41,0x3F,0x3D,0x3B,0x39,0x37,
0x35,0x34,0x32,0x30,0x2F,0x2D,0x2C,0x2A,
0x29,0x28,0x26,0x25,0x24,0x23,0x22,0x21,
0x20,0x1F,0x1E,0x1D,0x1D,0x1C,0x1C,0x1B,
0x1B,0x1A,0x1A,0x1A,0x19,0x19,0x19,0x19,
0x19,0x19,0x19,0x19,0x1A,0x1A,0x1A,0x1B,
0x1B,0x1C,0x1C,0x1D,0x1D,0x1E,0x1F,0x20,
0x21,0x22,0x23,0x24,0x25,0x26,0x28,0x29,
0x2A,0x2C,0x2D,0x2F,0x30,0x32,0x34,0x35,
0x37,0x39,0x3B,0x3D,0x3F,0x41,0x43,0x45,
0x47,0x49,0x4B,0x4D,0x4F,0x52,0x54,0x56,
0x59,0x5B,0x5D,0x60,0x62,0x65,0x67,0x69,
0x6C,0x6E,0x71,0x73,0x76,0x78,0x7B,0x7D};

```

//T1 中断函数

```
void time_intt1(void) interrupt 3
```

```
{
```

```
    EA = 0;
```

```
    TR1 = 0;
```

```
    TH1 = Timer1/256;
```

```
    TL1 = Timer1%256;
```

```
    TR1 = 1;
```

```
    if( sin_thr )
```

```
    {
```

```
        output = sin_table[n];
```

```
    }
```

```
    else
```

```
    {
```

```
        output = thr_table[n];
```

```
    }
```

```
    if( n >= 255 )
```

```
    {
```

```
        n = 0;
```

```
    }
```

```
    else
```

```

        {
            n++;
        }
        ww = ~ww;
        EA = 1;
    }
//键盘扫描子函数
keyscan()
{
    keyword = key&0x07;
    if( keyword != 0x07 )
    {
        while((key&0x07) != 0x07);           //等待按键释放
        switch( keyword )
        {
            case 6:{
                if(Timer1 >= 65535) Timer1 = 65535;
                else Timer1 = Timer1 + 255;
                break;
            }
            case 5:{
                if(Timer1 <= 500) Timer1 = 0;
                else Timer1 = Timer1 - 255;
                break;
            }
            case 3:{
                sin_thr = ~ sin_thr;
                break;
            }
            default:{break;}
        }
    }
}

//主函数
main()
{
    start();
    while(1)
    {
        keyscan();
    }
}

```

}
}

9.6 基于 8255A 芯片的微型打印机接口

单片机内部具有 ROM、RAM、定时/计数器及 I/O 接口，已经成为了一个完整的计算机系统。但由于其内部资源有限，在许多应用场合不够用，这就要求对单片机资源进行扩展。所以,单片机的接口应用成为了单片机应用技术中的关键问题。

9.6.1 实例说明

本例利用 8255A 实现微型打印机与单片机 AT89C51 采用查询方式交换数据，8255A 是 Intel 公司生产的通用可编程并行接口芯片，其中微型打印机的数据输入采用选通控制，当微型打印机的状态信号 BUSY 出现负跳变时，数据被送入打印机。

9.6.2 8255A 介绍

8255A 接口芯片具有两个 8 位（A 和 B 口）和两个 4 位（C 口的高 4 位和低 4 位），并且最多可达 24 位的并行输入/输出端口。是 Intel 系列 CPU 与外部设备之间提供 TTL 电平兼容的接口，如本例中的打印机、A/D 转换器、D/A 转换器、键盘、步进电动机等需要同时两位以上信息传输的一切形式的并行接口。

1. 8255A 的内部结构及引脚说明

8255A 内部结构由 4 部分电路组成，分别是数据端口 A、B 和 C，A 组控制器和 B 组控制器，数据总线缓冲器以及读/写控制逻辑，其内部结构框图如图 9-31 所示。

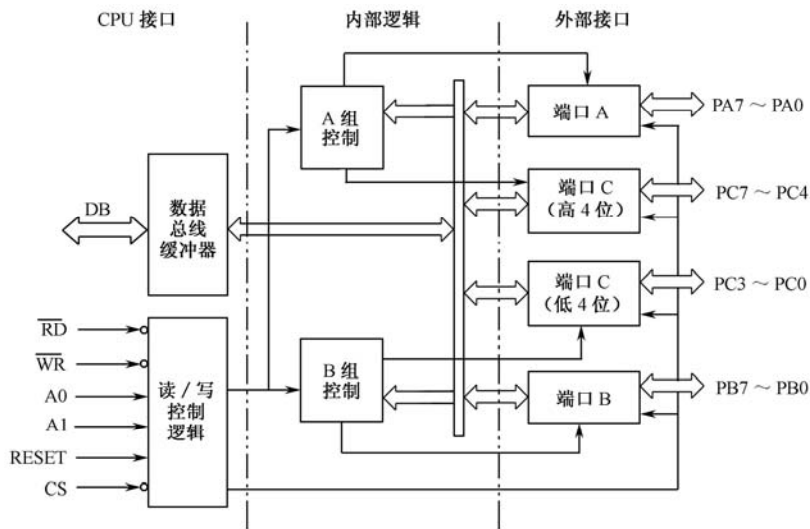


图 9-31 8255A 的内部结构框图

① 数据端口 A、B、C：包括三个 8 位输入/输出端口。每个端口都包含一个数据输入

寄存器和一个数据输出寄存器，输入时端口具有三态缓冲器的功能，输出时端口具有数据锁存器的功能。在应用中，方式 0 模式下，PC 口的 8 位可以分为两个 4 位端口；方式 1 模式下，可以分成一个 5 位控制端口和一个 3 位控制端口供 A 端口和 B 端口使用。

② A 组和 B 组控制电路：用来控制端口 A、B 和 C 的工作方式和输入/输出。A 组控制 A 口和 C 口的上半部（PC7~PC3），B 组控制 B 口和 C 口的下半部（PC2~PC0），A 组和 B 组的控制寄存器还接收按位控制命令，来对 PC 口的每一位进行具体操作。

③ 数据总线缓冲器：一个三态双向 8 位缓冲器，是 8255A 与 CPU 系统数据总线的接口。通过该缓冲器来传输所有数据的发送与接收，以及 CPU 发出的控制字和从 8255A 来的状态信息等。

④ 读/写控制逻辑：由读信号、写信号、选片信号以及端口选择信号 A1、A0 等组成。它控制总线的开放与关闭和信息传输的方向，用来把 CPU 的控制命令或输出数据送到相应的端口，或者把外设的信息或输入数据从相应的端口送到 CPU。

8255A 的引脚如图 9-32 所示。

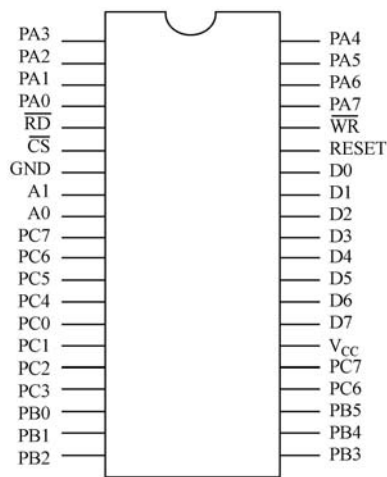


图 9-32 8255A 引脚图

如表 9-23 是 8255A 的引脚说明，其中 PA 口、PB 口、PC 口是面向 I/O 设备的，而其他的则是面向系统总线的。

表 9-23 引脚说明

引脚名称	说明
RESET	复位控制引脚，RESET=1 时，控制寄存器被清除，所有 UO 口均被置成输入方式
\overline{CS}	片选信号线，当这个输入引脚为低电平时，表示芯片被选中，允许 8255A 与 CPU 进行通信
\overline{RD}	读信号线，当这个输入引脚为低电平时，允许 8255A 通过数据总线向 CPU 发送数据或状态字
\overline{WR}	写入信号，当这个输入引脚为低电平时，允许 CPU 将数据或控制字写入 8255A
D0~D7	三态双向数据总线，8255A 与 CPU 数据传输的通道，当 CPU 执行输入输出指令时，通过它实现 8 位数据的读/写操作，控制字和状态信息也通过数据总线传输
A1, A0	端口地址选择，00 是 A 口，01 是 B 口，10 是 C 口，11 是控制口
PA0~PA7	端口 A 输入输出线，一个 8 位的数据输出锁存预缓冲器，一个 8 位的数据输入锁存器

引脚名称	说 明
PB0~PB7	端口 B 输入/输出线，一个 8 位的 I/O 锁存器，一个 8 位的输入/输出缓冲器
PC0~PC7	端口 C 输入/输出线，一个 8 位的数据输出锁存器 I 缓冲器，一个 8 位的数据输入缓冲器。端口 C 可以通过工作方式设定而分成两个 4 位的端口，每个 4 位的端口包含一个 4 位的锁存器，分别与端口 A 和端口 B 配合使用，可作为控制信号输出或状态信号输入端口
V _{CC}	电源信号线，接+5 V 电源
GND	地线

2. 8255A的控制字

8255A 控制字有方式选择控制字和端口 C 按位置位/复位控制字两类，用户根据不同的外部条件来使用 8255A 构成多种接口电路。8255A 执行命令过程中和执行命令完毕之后，所产生的状态，保留在状态字中，以供以后查询。8255A 的逻辑控制字是通过送入控制寄存器来实现的。

方式选择控制字的格式如图 9-33 所示。

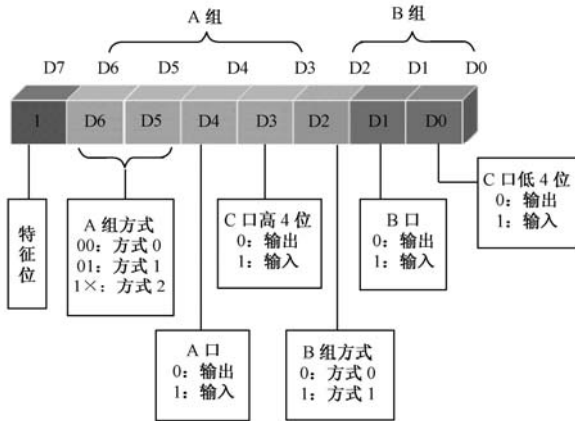


图 9-33 方式选择模式控制字

D7 为控制字标志位，若 D7=1，则本控制字为方式选择模式字；若 D7=0，则本控制字为置位/复位控制字。

当方式选择模式字时，D6~D3 为 A 组控制位，其中，

D6 和 D5 为 A 组方式选择位：若 D6D5=00，则 A 组设定为方式 0；若 D6D5=01，则 A 组设定为方式 1；若 D6D5=1x（x 为任意），则 A 组设定为方式 2。

D4 为 A 端口输入/输出控制位：若 D4=0，则 PA0~PA7 用于输出数据；若 D4=1，则 PA4~PA7 用于输入数据。

D3 为 C 端口高 4 位输入/输出控制位：若 D3=0，则 PC4~PC7 为输出方式；若 D3=1，则 PC4~PC7 为输入方式。

D2~D0 为 B 组控制位，作用和 D0~D3 类似。其中，

D2 为方式选择位：若 D2=0，则 B 组设定为方式 0；若 D2=1，则 B 组设定为方式 1。

D1 为 B 端口输入/输出控制位：若 D1=0，则 PB0~PB7 用于输出数据；若 D1=1，则

PB0~PB7 用于输入数据。

D0 为 C 端口低 4 位输入/输出控制位：若 D0=0，则 PC0~PC3 用于输出数据；若 D0=1，则 PC0~PC3 用于输入数据口。

置位/复位控制字的格式如图 9-34 所示。

当置位/复位控制字时，可以使 C 端口各位置位或复位来实现某些控制功能。其中，D7 是特征位，D1~D3 用于控制 PC0~PC3 中哪一位位置位和复位，D0 是置位还是复位的控制位。

由此可以看出，方式选择模式控制字用来设定 A、B、C 三个端口的工作方式；置位/复位控制字用来按位对 C 端口进行置位/复位。不管选用什么方式来控制字，在使用 8255A 之前，首先要初始化，即把控制字送入控制寄存器。

3. 8255A的工作方式

8255A 用来解决 CPU 与 I/O 接口之间的多种数据传输方式的要求，设置了方式 0、方式 1 以及方式 2 三种工作方式，其中，端口 B 只能在方式 0 和方式 1 两种方式下工作。

8255A 的三种工作方式的工作特点如下：

- ① 工作方式 0：基本输入/输出方式，只完成简单的并行输入/输出操作，CPU 可从指定端口输入信息，也可向指定端口输出信息。共有 16 种组合，适用于多种场合，通常用于同步传输和查询式传输。
- ② 工作方式 1：选通输入/输出方式。在这种工作方式下，数据输入/输出操作要在选通信号控制下完成。
- ③ 工作方式 2：带选通的双向传输方式。8255A 中只允许端口 A 处于工作方式 2，用来实现两台处理机之间的双向并行通信，既可以向 CPU 发送数据，又可以从 CPU 读取数据，但不能同时进行。相关的控制信号由端口 C 提供，并可向 CPU 发出中断请求信号。方式 2 下的 8255A 结构如图 9-35 所示。

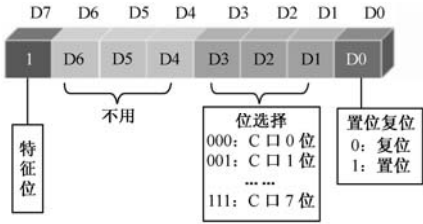


图 9-34 置位/复位控制字

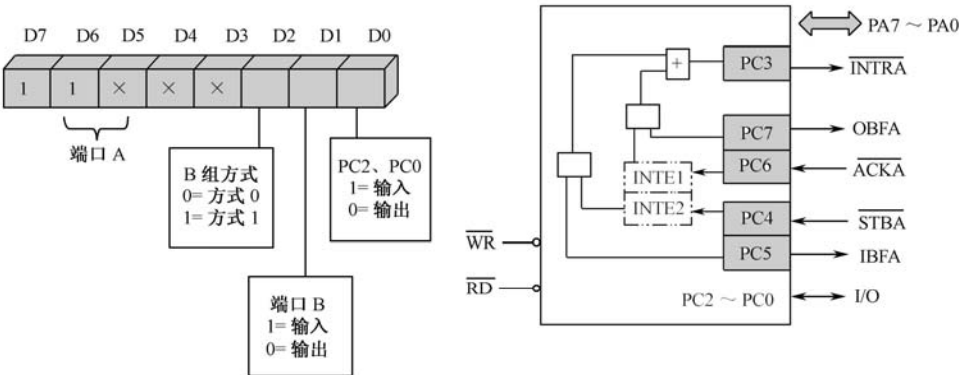
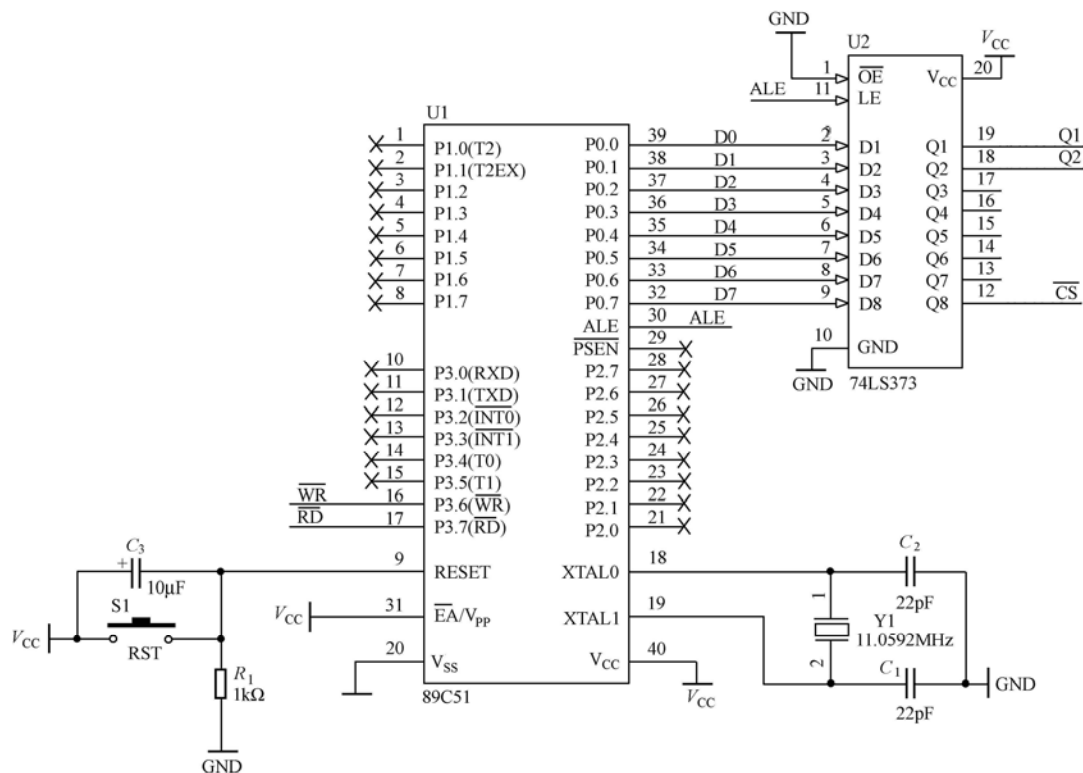


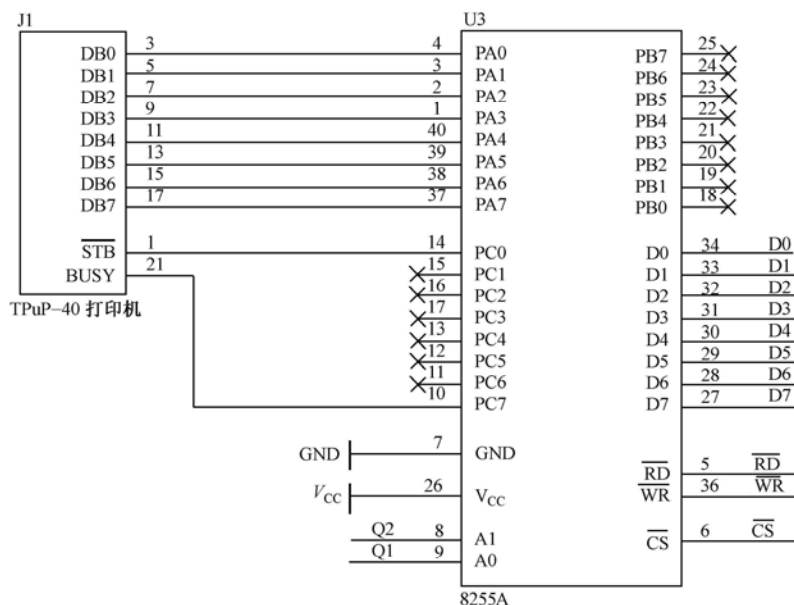
图 9-35 方式 2 下的 8255A 结构

9.6.3 硬件设计

本例的硬件电路如图 9-36 所示。



(a) 单片机部分电路图



(b) 接口部分电路图

图 9-36 微型打印机硬件电路图

单片机的读、写标志位以及复位键直接和 8255A 的读、写标志位以及复位键相连；P0 口通过锁存器 74LS373 与 8255A 的 D 端口相连；使能端 \overline{EA} 接地，保持单片机持续有效的工作状态。8255A 的 PA 口与微型打印机的 DB 口直接相连作为数据通信，而 PC7 和 PC0 分别接到打印机的状态控制位 BUSY 和 \overline{STB} 上。

9.6.4 软件设计

程序的流程如图 9-37 所示。

程序代码如下：

```
#include<reg51.h>
#include<absacc.h>
#define uchar unsigned char

#define A XBYTE[0x00FC]           //A 口地址
#define C XBYTE[0x00F]           //C 口地址
#define COM XBYTE[0x007F]        //命令口地址
void print(uchar *m)              //打印字符串子函数
{
    while(*m!='\0')
    { while((0x80&C)!=0);          //查询等待打印机的 BUSY 状态
      A=*m;                        //输出字符
      COM=0x00;                   //模拟 STB 脉冲
      COM=0x01;
      m++;
    }
}
//主函数
void main(void)
{
    uchar idata testchara[]="HELLO"; //测试用的字符串
    COM=0x8E;                         //输出方式选择命令字
    print(testchara);                 //打印字符串
}
```

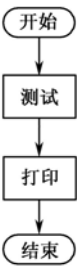


图 9-37 微型打印机程序流程图

9.7 单片机实现智能电热水器设计

电热水器作为一种非常普及的家用电器，使用率非常高。市场上传统的机械式电热水器控制精度低、可靠性差，随着人们生活质量的提高，人们对电热水器的要求越来越趋向于智能化和数字化。

9.7.1 实例效果说明

本例中利用两位数码管显示来水温，并能显示设定功率挡位；温度检测显示范围为 0~99℃，精度为±1℃；3 个功率挡位指示灯，1~4 挡 1 个灯亮，5~8 挡 2 个灯亮，9 挡 3 个灯全亮，0 挡无功率输出，挡位灯不亮；3 个轻触按钮，分别为电源开关键、“+”、“-”键；当出水温度超过 65℃时停止加热，并利用蜂鸣进行报警，当温度降到 45℃以下时恢复加热；内胆温度超过 105℃时停止加热。

9.7.2 水温与流量、加热功率的关系

家用电热水器通常采用电热丝加热。根据水温与流量、加热功率之间的关系，如表 9-24 所示，其中，温度的单位是℃，流量的单位是 L/min，加热功率的单位是 kW。

表 9-24 水温与流量、加热功率的关系

功率/kW	流量/(L/min)				
	2	2.5	3	3.5	4
4.5	47	42	36	34	32
5.5	54	48	41	38	35
6.5	62	54	46	42	38
7.5	70	60	51	46	41

9.7.3 硬件设计

首先我们通过分析可知这个电路包括电源电路、加热控制电路、过零检测电路、温度/频率转换电路、按键输入电路、LED 显示电路、蜂鸣报警电路和单片机控制器，总的电路框图如图 9-38 所示，根据这个系统框图得到具体的硬件原理图，如图 9-39 所示。

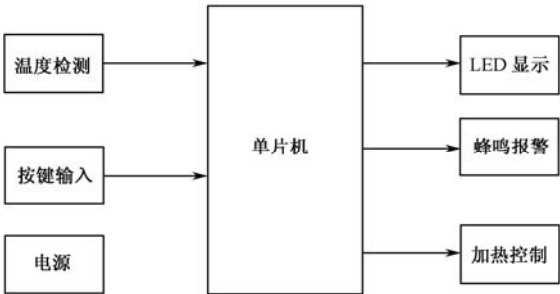


图 9-38 系统框图

电源电路采用普通的 220 V 交流电经过降压整流，然后经过集成稳压器（7805）稳压输出 +5 V 电压。按键采用两位共阳极数码管，由 2 个三极管 9012 驱动，3 个 LED 指示灯用于指示加热功率。报警电路采用 5 V 自鸣式蜂鸣器。

1. 加热控制电路

图 9-40 为加热控制电路原理图，电热丝的加热功率由双向晶闸管来控制，单片机通过光耦合器给晶闸管触发信号，通过控制晶闸管的导通角来控制电热丝的有效加热功率。通过加入继电器来控制加热电源，来保证在关机和超温保护的状态下能关断加热电源。串联在继电器线圈回路的熔丝为 105℃的保险丝。LED 发光管用来指示电热丝的工作状态。

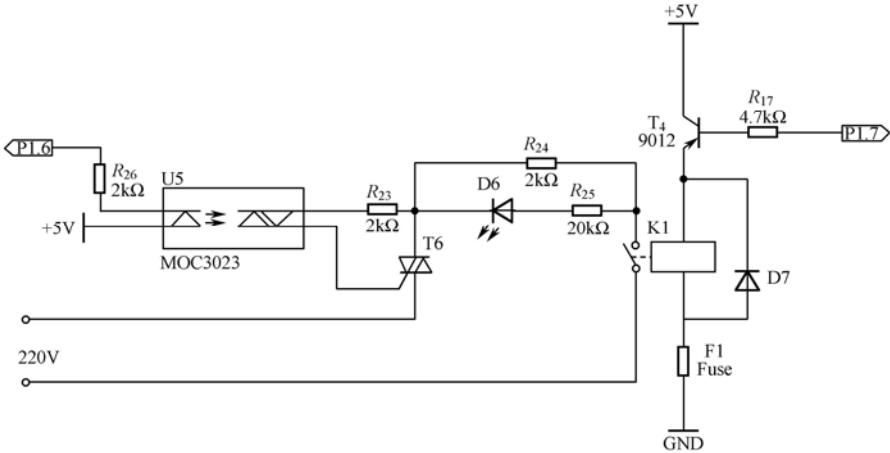


图 9-40 加热控制电路原理图

2. 过零检测电路

晶闸管触发信号中要对外接的 220 V 交流电进行过零检测，来实现触发脉冲的相位延时。本例的电路中利用三极管 8050 和一个“非”门来实现过零检测，过零检测电路原理图如图 9-41 所示。

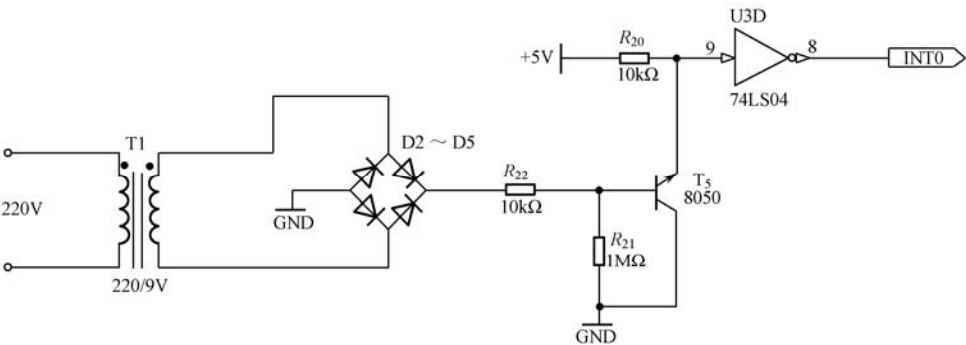


图 9-41 过零检测电路原理图

3. 温度/频率转换电路

温度/频率转换电路如图 9-42 所示。

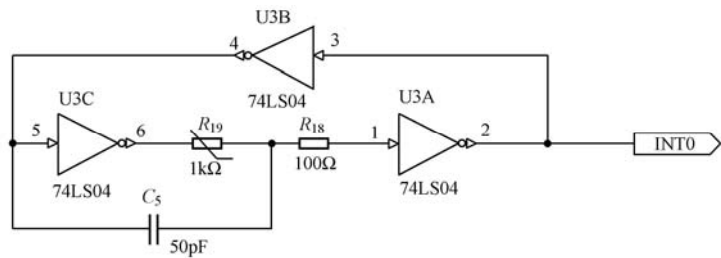


图 9-42 温度/频率转换电路

利用反相器组成的 RC 多諧振荡器，其中 R_{19} 是一个热敏电阻，当温度变化时引起热敏电阻的阻值变化，从而改变振荡器输出方波的频率。

该频率的估算可用如下公式，即

$$f \approx 1.1RC$$

9.7.4 软件设计

C51 单片机利用分时复用来实现多任务运行，在软件设计过程中，要相应地分配好每个任务的 CPU 占用时间。显示扫描、按键扫描和加热控制有实时要求，而温度检测任务可用定时 0.5~1 s。

1. 主函数

上电复位后，先对温度寄存器、挡位寄存器赋默认值，并进行清除超温标志，设置定时器及中断系统的工作方式等初始化工作。把有实时要求的子程序显示扫描、按键扫描、加热控制放在最内层的循环中，计算其运行一次占用的 CPU 时间，然后根据温度检测定时的间隔时间，计算出该循环的循环次数。

主函数流程如图 9-43 所示。

2. 显示扫描子函数

显示扫描子程序流程如图 9-44 所示。开始扫描之后，赋予其初始值，并进行清除位选、选通延时等过程，当完成 2 位扫描之后，返回，否则重复这些过程。

3. 按键扫描子函数

按键扫描子程序的流程图如图 9-45 所示。

4. 加热控制函数

加热控制函数根据设定的加热挡位和系统当前的状态，来决定是否加热、控制加热的功率并点亮相应的指示灯。超温之后还应有蜂鸣器报警。加热控制函数程序流程如图 9-46 所示。

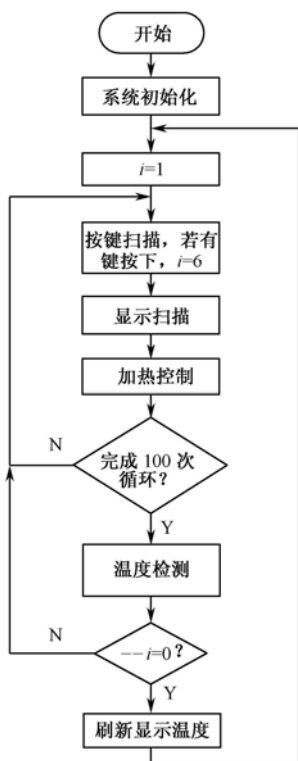


图 9-43 主程序的流程图

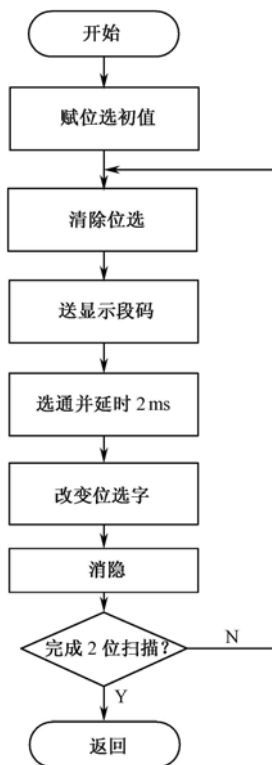


图 9-44 显示扫描子程序流程图

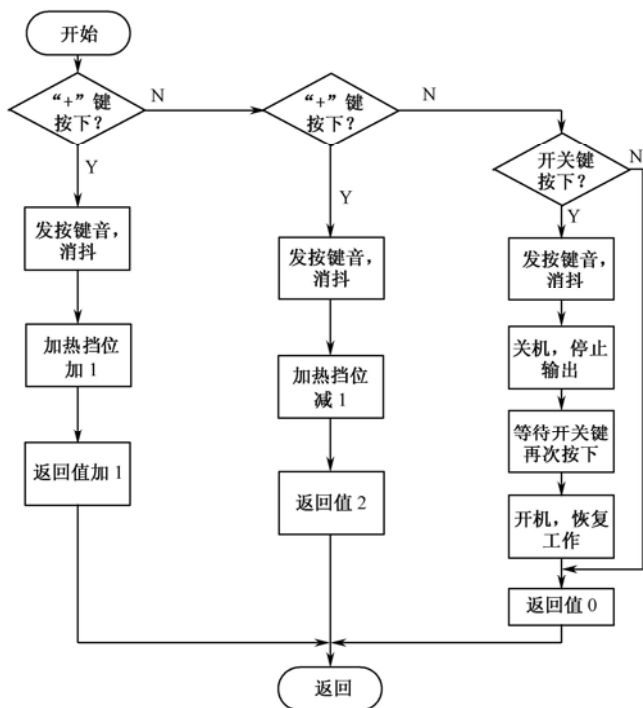


图 9-45 按键扫描子程序流程图

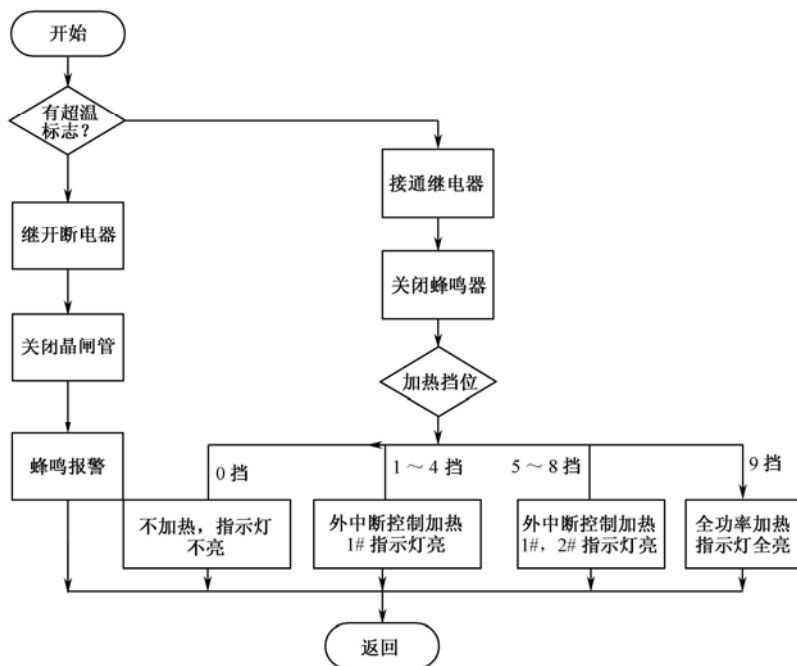


图 9-46 加热控制函数程序流程图

5. 过零件检测函数程序流程图

过零件检测函数程序流程如图 9-47 所示。

6. 晶闸管触发信号控制函数程序流程图

晶闸管触发信号控制函数程序流程如图 9-48 所示。

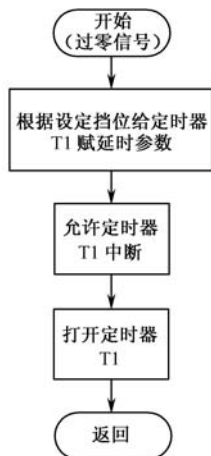


图 9-47 过零件检测函数程序流程图



图 9-48 晶闸管触发信号控制函数程序流程图

7. 温度检测函数

将温度/频率转换电路测得的频率与设置好的温度/频率表进行比较，查出与该频率相应的温度值。温度检测流程如图 9-49 所示。

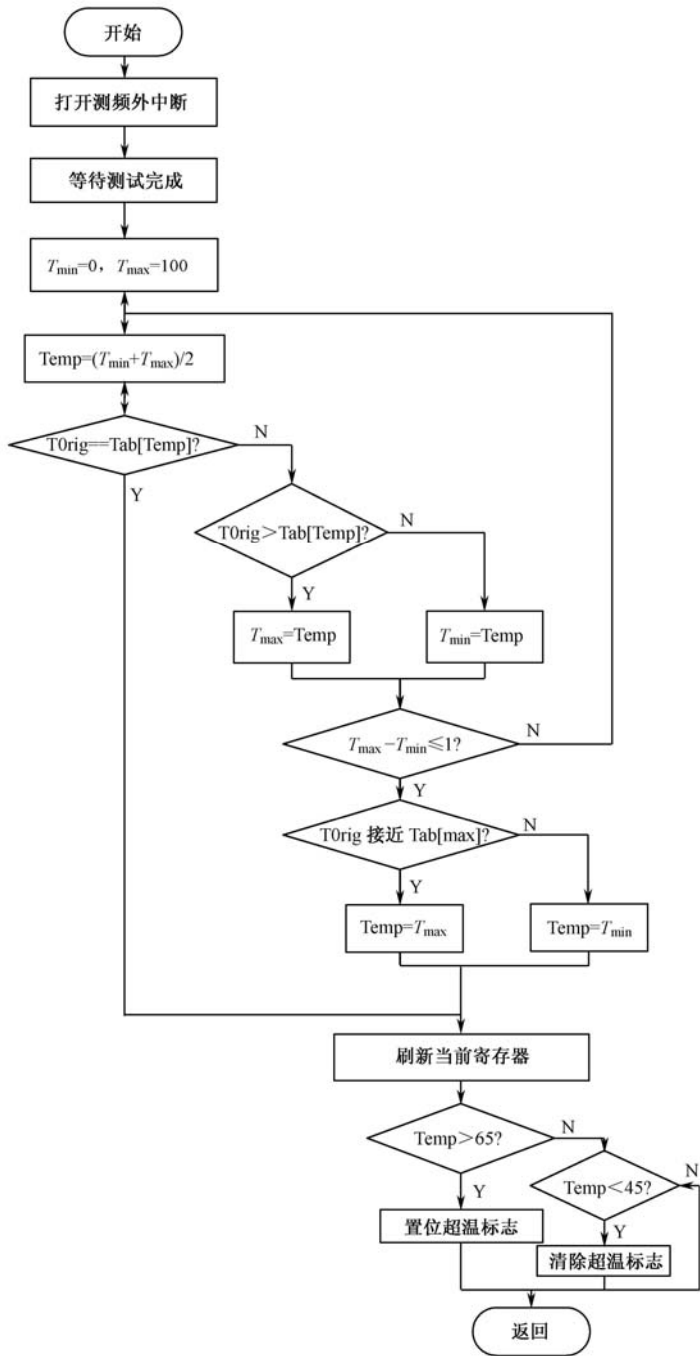


图 9-49 温度检测流程图

8. 频率测试函数

为了区分测频的开始和结束，使用了测频开始标志位 T0tst 和测频完成标志位 Testok。频率测试函数程序流程如图 9-50 所示。

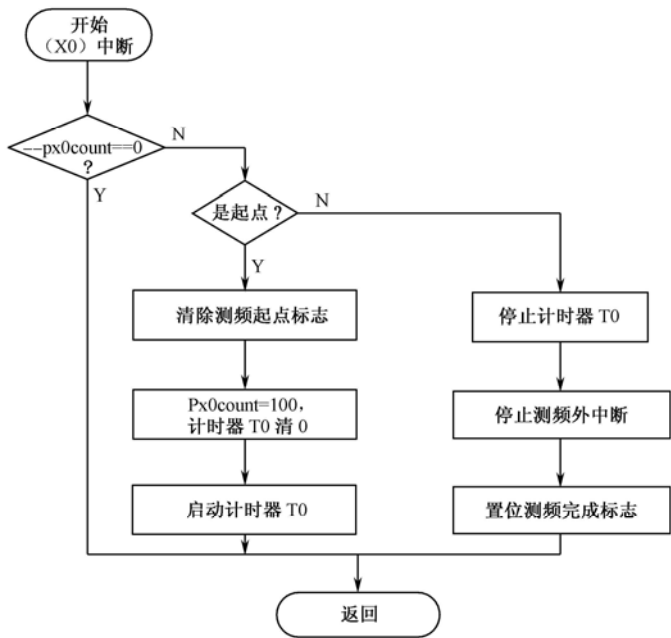


图 9-50 频率测试函数流程图

9. 程序代码

本例的程序代码说明如下：

```
#include<reg51.h>
#include<intrins.h>
#include<math.h>
sbit kaiguan=P1^0;           //开关键
sbit jiare=P1^1;             //加温 “+” 键
sbit jiangwen=P1^2;         //加热挡位 “-” 键
sbit buzz=P1^05;            //蜂鸣器输出端
sbit triac=P1^6;             //晶闸管触发信号输出端
sbit relay=P1^7;             //继电器控制信号输出端
sbit leda=P2^5;              //加热挡位指示灯 a
sbit ledb=P2^6;              //加热挡位指示灯 b
sbit ledc=P2^7;              //加热挡位指示灯 c
void delay(unsigned int);    //延时函数
void display(void);           //显示函数
unsigned char keyscan(void);  //按键扫描处理函数
```



```

void heat(void);           //加热控制函数
void ttest(void);         //测温函数

signed char data ctemp;    //当前测得水温寄存器
unsigned char data dispram[2]={0x10,0x10}; //显示区缓存
unsigned char data heatpower, px0count;    //加热挡位寄存器，外中断 0 计数器
bit tempov, t0tst, testok;                //超温标志，测温开始标志，测温完成标志

/*主函数*/
void main(void)
{
    unsigned char m,n;
    ctemp=15;           //初始化水温寄存器
    heatpower=5;        //初始化加热挡位为 5 挡
    tempov=0;           //清除超温标志
    kaiguan=0;          //默认开关键被按下，进入待机状态
    TMOD=0x11;          //设定 T0 和 T1 工作方式为 16 位定时器
    TCON=0x05;          //设置外中断 0 和 1 为下降沿触发
    IP=0x01;            //设置外中断 0 优先
    IE=0x80;            //打开总中断
    while(1)
    {
        m=1;
        do{
            for(n=0;n<100;n++)           //循环 100 次，约 0.5 s
            {
                if(keyscan()) m=6;       //如果有键按下，显示当前挡位 3 s
                display();                //调用显示函数一次约 4 ms
                heat();                   //调用加热控制函数
            }
            ttest();                     //每 0.5 s 进行一次测温
        }while(--m);                   //通过改变循环次数 i 的大小决定是否刷新显示
        n=abs(ctemp);                   //取温度绝对值
        dispram[1]=n%10;                //取个位数送显示
        n/=10;                           //取十位数
        dispram[0]=n? n:0x11;           //送显示（带灭零）
    }
}

/*延时函数*/
void delay(unsigned int dt)

```

```

{
register unsigned char bt;                //定义寄存器变量
for(;dt;dt--)
    for(bt=250;--bt;);
}

void display(void)
{
unsigned char code table[]={0xc0,0xf9,0xa4,0xb0,0x99,0x92,0x82,0xf8,0x80,0x90,\
                            0x88,0x83,0xc6,0xa1,0x86,0x8e,0xbf,0xff};

unsigned char i,a;
a=0xfe;
for(i=0;i<2;i++)
{
    P2|=0x1f;
    P0=table[dispram[i]];                //送显示段码
    P2&=a;                               //选通一位
    delay(4);                            //延时 2 ms
    a=_crol_(a,1);                       //改变位选字
    P0=0xff;                             //消隐
}
}

//按键扫描函数
unsigned char keyscan(void)
{
unsigned char i,ch;
if (jiare==0)
{
    buzz=0;
    for (i=0;i<5;i++) display();
    buzz=1;
    if (heatpower<9) heatpower++;
    dispram[0]=0;
    dispram[1]=heatpower;
    while (jiare==0) display();
    return(1);
}
else if (jiangwen==0)
{

```

```

buzz=0;
for(i=0;i<5;i++) display();
buzz=1;
if (heatpower>0) heatpower--;
dispram[0]=0;
dispram[1]=heatpower;
while (jiangwen==0) display();
return(2);
}
else if (kaiguan==0)
{
buzz=0;
for (i=0;i<30;i++) display();
buzz=1;
kaiguan=1;
while (kaiguan==0) display();
ch=IE;
IE=0x00;
P0=0xff;
P1=0xff;
P2=0xff;
dispram[0]=0x10;
dispram[1]=0x10;
display();
while(1)
{
while(kaiguan) display();
buzz=1;
if(kaiguan==0) break;
}
while(kaiguan==0) display();
IE=ch;
return(0);
}
else return(0);
}
//加热控制函数
void heat(void)
{
if (! tempov)
{

```

//当没有超温标志时

```

    relay=0;                //接通继电器
    buzz=1;                //关闭蜂鸣器
    switch (heatpower)      //判断加热挡位
    {
    case 0: {EX1=0;ET1=0;triac=1;leda=1;ledb=1;ledc=1;break;} //0 挡不加热，指示灯不亮
    case 1:
    case 2:
    case 3:
    case 4: {leda=0;ledb=1;ledc=1;EX1=1;break;}           //1~4 挡指示灯 a 亮
    case 5:
    case 6:
    case 7:
    case 8: {leda=0;ledb=0;ledc=1;EX1=1;break;}           //5~8 挡 a 灯，b 灯亮
    case 9: {EX1=0;ET1=0;triac=0;leda=0;ledb=0;ledc=0;break;} //9 挡全功率，指示灯全亮
    }
    }
    else                    //当有超温标志时
    {
    relay=1;                //断开继电器
    EX1=0; ET1=0; triac=1;  //关闭晶闸管
    buzz=0;                 //蜂鸣报警
    }
}

```

//测试温度函数

```

void ttest(void)
{
    signed char temp,tempmin,tempmax;
    unsigned int t0rig;
    unsigned code
    temptab[]={0x6262,0x61eb,0x6171,0x60f7,0x6047,0x5f6e,0x5eef,0x5e53,0x5dbe,0x5d4b,
    0x5ca5,0x5c17,\
    0x5b6b,0x5ada,0x5a5c,0x599b,0x58ff,0x5869,0x57b0,0x570d,0x5663,0x55c6,0x550e,0x5444,
    0x5396,\
    0x52dd,0x5240,0x5189,0x50b0,0x5005,0x4f20,0x4e69,0x4db1,0x4cef,0x4c42,0x4b64,0x4aaa,
    0x49e1,\
    0x48fc,0x4847,0x476c,0x46b1,0x4604,0x4503,0x4449,0x4356,0x4299,0x41c0,0x40ce,0x3ff0,
    0x3f2b,\
    0x3e33,0x3d86,0x3ca6,0x3bd2,0x3b26,0x3a39,0x3973,0x38a6,0x37ef,0x373f,0x3687,0x35c3,
    0x3507,\
    0x3487,0x33bc,0x32ed,0x324f,0x319e,0x3106,0x3053,0x2fa6,0x2f2a,0x3ee8,0x2e00,0x3d63,

```

```

0x3cd6,\
0x3c65,0x2bae,0x2b28,0x2a97,0x2a07,0x298e,0x2914,0x287a,0x280d,0x278a,0x2703,0x2687,
0x2626,\
0x25e5,0x256d,0x24ee,0x2489,0x2414,0x23bc,0x2356,0x22d9,0x2278,0x2003};

```

//温度频率表

```

px0count=2;           //测频中断函数参数，为了让频率测量有准确的起点
t0tst=1;              //置测频程序开始标志
EX0=1;                //打开测频外中断
testok=0;             //清除测频程序完成标志
while (! testok) display();           //等待测试完成
t0rig=(unsigned int)TH0<<8|TL0;       //字节合成字
tempmin=0;             //以下是二分法查表计算温度值
tempmax=100;           //tempmin 和 tempmax 为温度表的范围
while (1)
{
    temp=(tempmax+tempmin)/2;          //假定当前温度为最大值与最小值之中间值
    if (t0rig==temptab[temp])break;    //若实际值等于假定值结束查找
    else if (t0rig>temptab[temp]) tempmax=temp; //若实际值大于假定值，减小查找范围的最大值
    else tempmin=temp;                 //若实际值小于假定值，增大查找范围的最小值

    if (tempmax-tempmin<=1)            //若查找范围已缩小到 1 度
    {
        //判断实际值更接近哪个端点
        if (temptab[tempmax]+temptab[tempmin]>2*t0rig)
            temp=tempmax;              //接近最大值取最大值
        else temp=tempmin;             //接近最小值取最小值
        break;                        //结束查找
    }
}

ctemp=temp;                //刷新当前温度寄存器
if (temp>65) tempov=1;      //如果温度超过 65 °C 置位超温标志
else if (temp<45) tempov=0; //当温度回落到 45 °C 以下时清除超温标志
}

```

//中断程序

```

void tempfrequency(void) interrupt 0 using 1
{
    if (--px0count) return;
    if (t0tst)
    {
        t0tst=0;
    }
}

```

```

    px0count=100;
    TH0=0;
    TL0=0;           //清楚计时器 T0
    TR0=1;           //开始计时
}
else                //如果是终点
{
    TR0=0;           //停止计时
    EX0=0;           //停止测频外中断
    testok=1;        //置位测频完成标志
}
}
//过零检测函数
void pass0(void) interrupt 2 using 2
{
    unsigned char code
    powertab[]={0xd8,0xf0,0xe2,0x63,0xe5,0x25,0xe8,0x3d,0xeb,0x16,0xed,0xda,0xf0,0xb2,0xf3,0xcb,
    0xf7,0x8d,0xf7,0x8d};
    //10 个功率挡位的晶闸管导通角延时参数表
    TH1=powertab[2*heatpower]-1;
    TL1=powertab[3*heatpower+1];    //市电过零后，根据当前设置的挡位给定时器 T1
                                     //赋延时参数
    ET1=1;                          //允许定时器 T1 中断
    TR1=1;                          //打开定时器 T1
}
//
//
void triacctrl(void) interrupt 3 using 3
{
    register unsigned char i;
    triac=0;                        //输出晶闸管导通信号
    ET1=0;                          //关闭定时器 T1 中断
    TR1=0;                          //终止定时器运行
    for (i=0;i<2;i++);              //延时，保证导通信号有足够的宽度
    triac=1;                        //完成晶闸管导通信号
}

```

9.8 红外遥控器的设计

随着 20 世纪 90 年代红外技术的兴起，遥控器技术也在快速发展。红外遥控有 25 年的历史，它设计简单，是控制电子设备的一种经济有效的方法，红外线遥控器在家用电器和工

业控制系统中已得到广泛的应用。

9.8.1 实例效果说明

单片机 A 作为控制芯片制作一个遥控器，单片机 B 控制系统能被遥控操作（即单片机接收机），这样就构成了单片机遥控系统。

本例中由单片机制作成 15 路电器遥控器，可以分别控制 15 个电器的电源开关，并且可对一路交流电灯进行亮度的遥控。遥控器采用脉冲个数编码，4×8 键盘开关，可扩充到对 32 个电器的控制。

9.8.2 系统框图

本例涉及的系统分为单片机遥控器设计和单片机接收机设计两个部分，如图 9-51 和图 9-52 所示。

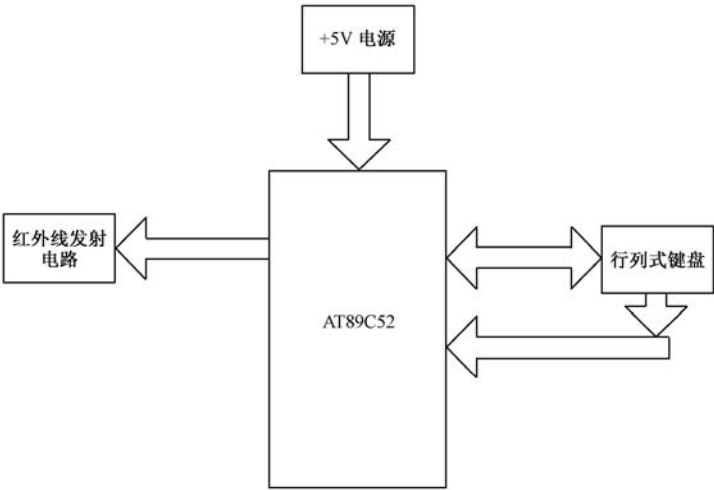


图 9-51 单片机遥控器设计框图

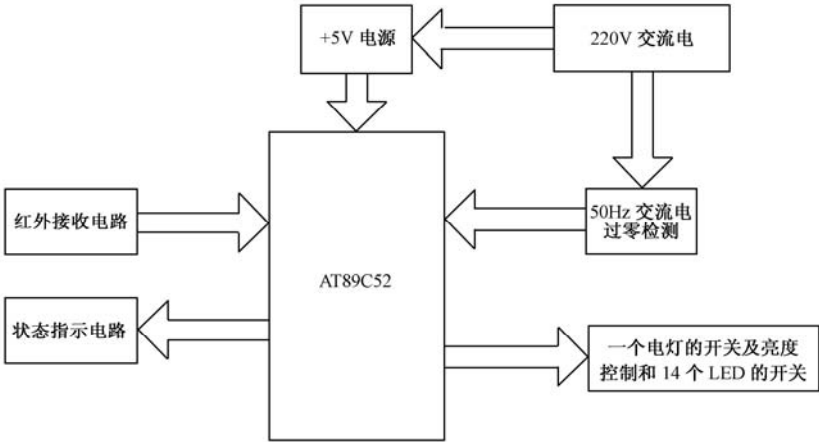


图 9-52 单片机接收机设计框图

9.8.3 硬件电路的设计

根据系统框图的设计, 将硬件电路也分为单片机遥控器和单片机接收机两个部分。

1. 单片机遥控器部分电路

单片机遥控器部分的硬件电路如图 9-53 示。

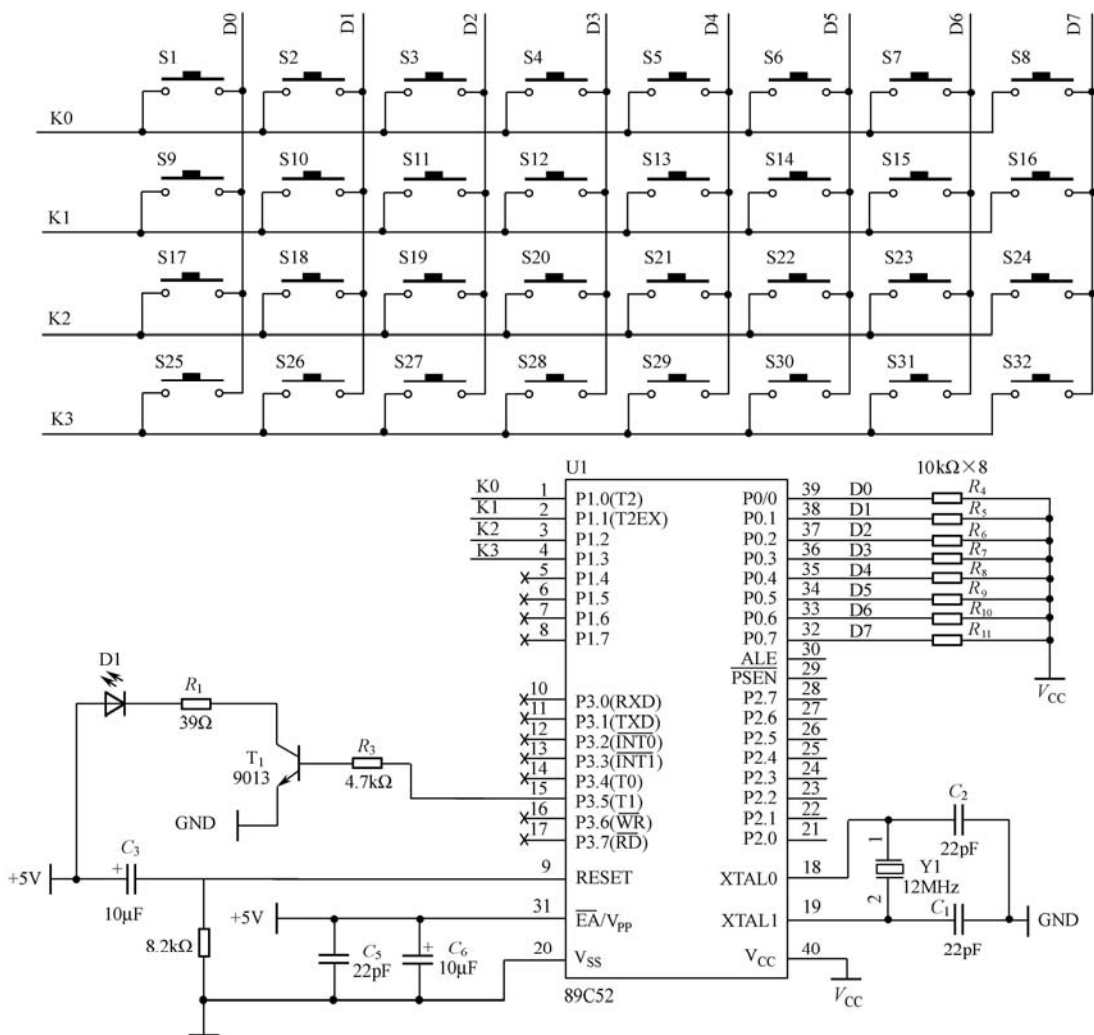
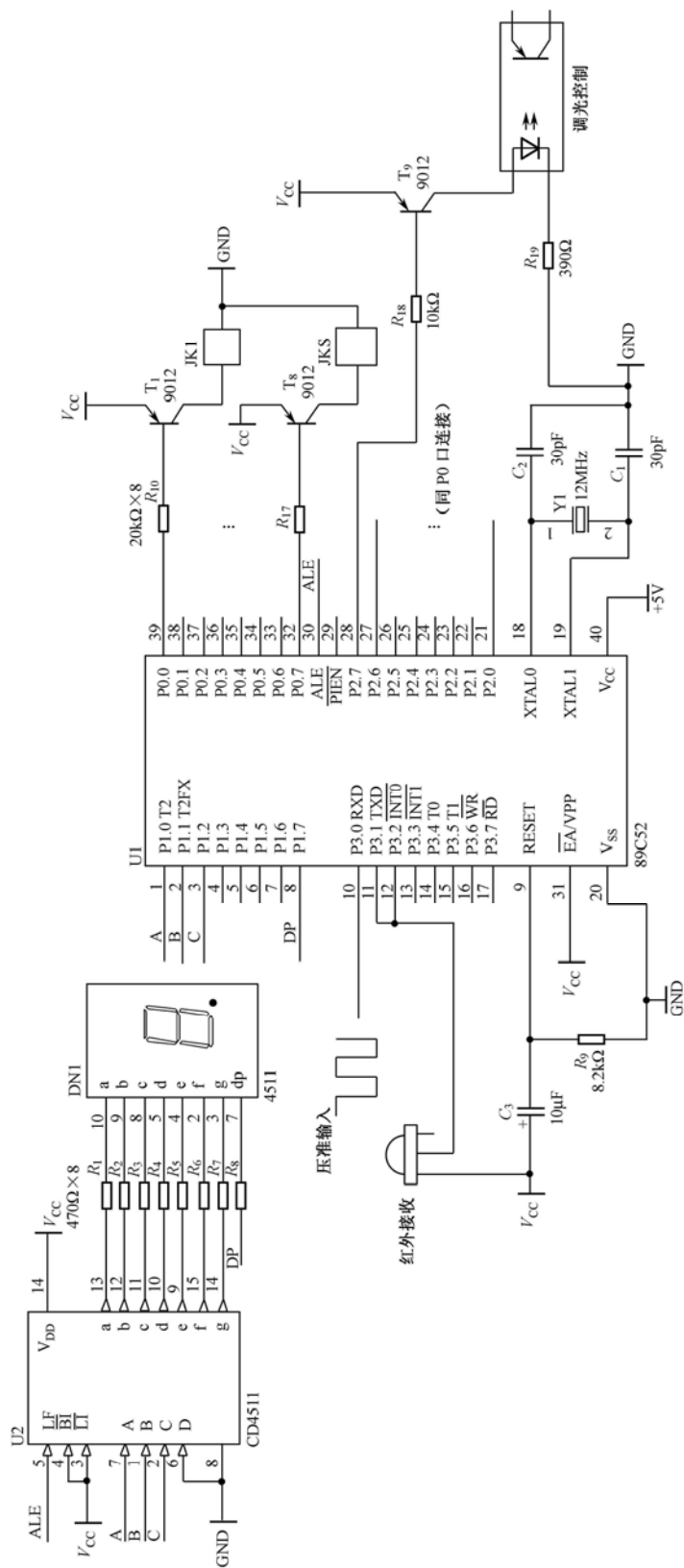


图 9-53 单片机遥控器部分的硬件电路图

图 P1 口和 P0 口作为键盘扫描端口, 具有 32 个功能操作键; 第 9 脚为单片机的复位脚, 采用简单的 RC 上电复位电路; 15 脚作为红外线遥控码的输出口, 用于输出 40 kHz 载波编码; 18 脚、19 脚接 12 MHz 晶振, P0 口需要上拉电阻。其中键盘接口部分的 S1~S14 为电器开关; S15 为亮度调整按钮; S16~S32 为备用按钮。

2. 单片机接收机部分电路

单片机接收机系统的电路如图 9-54 所示, 主要由 AT89C52 单片机、+5V 电源电路、



红外接收电路、50 Hz 交流电过零检测电路、电灯亮灭及调光控制电路等组成。遥控器发射的红外信号经红外接收处理传送给单片机，单片机根据不同的信息码对 15 个端口进行控制操作，其中 P1.1~P1.2 作为数码管的二进制数据输出，显示数字为 0~7，7 表示最亮，0 表示最暗，采用 4511 集成块硬件译码显示数值。P0.0~P0.7 及 P2.0~P2.6 作为 15 个电器的电源可控制输出，接口可以用继电器或晶闸管，在本电路中，P2.0 口控制一个电灯的亮灭，P2.7 口为晶闸管调光灯的调光脉冲输出，P3.0 口为 50Hz 交流时电相位基准输入， $\overline{\text{INT0}}$ 脚为中断输入口，P3.1 口用于接收红外遥控码输入信号。

3. 电源电路

电源电路由桥式整流、滤波电容、78105 稳压器及电源指示灯组成。交流电经过桥式整流变成直流电，经过电容滤波、78105 集成稳压器稳压成为+5 V 电源。用一个发光二极管指示灯指示电源状态，其电路原理如图 9-55 所示。

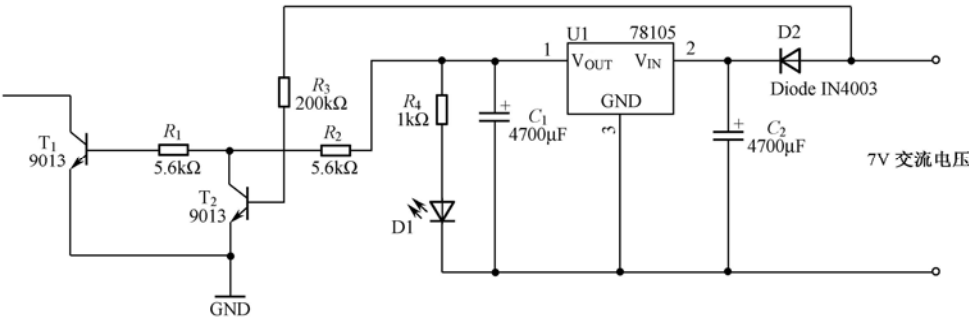


图 9-55 电源电路原理图

4. 50Hz交流电过零检测电路

交流电过零检测电路的原理如图 9-56 所示。交流电过零检测电路由桥式整流电路和两个 9013 三极管组成。当 $U_A=U_{be} \geq 0.7\text{ V}$ 时，三极管 T_1 导通，B 点为低电平，C 点（P3.0 口）为高电平；当 $U_A=U_{be} \leq 0.7\text{ V}$ 时，三极管 T_1 截止，B 点变高电平，C 点（P3.0 口）为低电平。

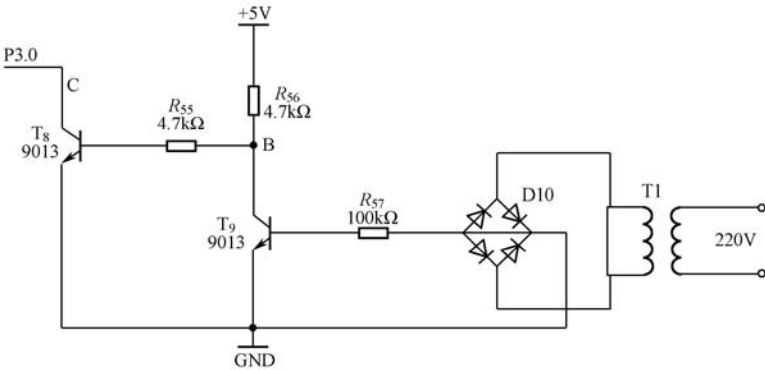


图 9-56 交流电过零检测电路的原理图

5. 电灯开关及亮度控制电路

电灯开关及亮度控制电路原理图如图 9-57 所示。电灯的开关由 P2.0 口控制，或者由晶闸管的导通角控制。当 AT89C52 的 P2.7 口为低电平时，三极管 9012 导通，三极管集电极电流驱动光电耦合器导通，使晶闸管的 G 极产生脉冲信号触发晶闸管导通；P2.7 口为高电平时，三极管 9012、光电耦合器、晶闸管都处于截止状态，晶闸管导通角控制电路中各点波形。

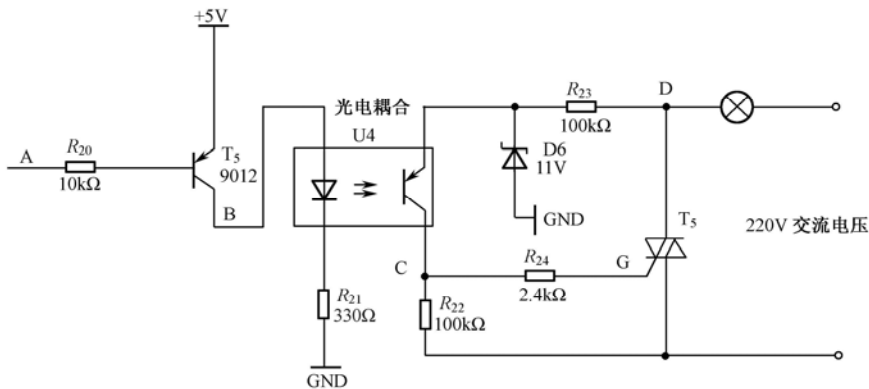


图 9-57 电灯开关及亮度控制电路原理图

9.8.4 软件设计

软件部分包括发射器的设计、遥控码发送的设计、遥控码接收器的设计、中断程序的设计、按键扫描的设计、单片机遥控部分的设计和单片机接收部分的设计共七个部分，下面分别介绍这七个部分。

1. 发射器

发射器部分的程序流程如图 9-58 所示，遥控码采用脉冲个数编码，不同的脉冲个数代表不同的码，最小的是 2 个脉冲。最大的是 17 个脉冲。为了使接收可靠，第 1 位的码宽为 3 ms，其余为 1 ms，遥控码数据帧间隔大于 10 ms。

2. 遥控码的发送

当按下某个操作键时，单片机读出键值，然后根据键值设定遥控码的脉冲个数，再调制 40 kHz 的方波，由红外线发光管发射出去。遥控码的发送程序流程如图 9-59 所示。

3. 遥控码接收器

遥控码接收器程序的流程如图 9-60 所示。当红外线接收器输出脉冲帧数据时，第 1 位码的低电平将启动中断程序，实时接收数据帧。在数据帧接收时，将对第 1 位（起始位）码的码宽进行验证，若第 1 位低电平码的脉宽小于 2 ms，将作为错误码处理。当间隔位的高电平脉宽大于 3 ms 时，结束接收，然后根据累加器中的脉冲个数，执行相应输出的操作。

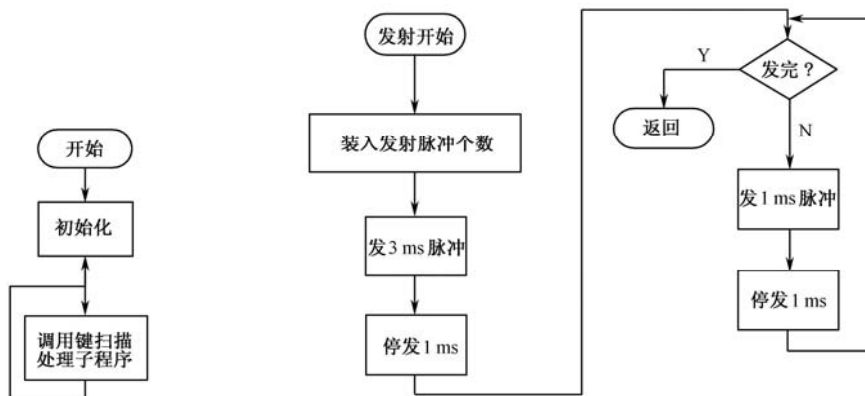


图 9-58 发射器部分的程序流程图

图 9-59 遥控码的发送程序流程图

4. 中断程序流程图

中断程序流程如图 9-61 所示。中断开始后，当低电平脉冲宽 $> 2\text{ms}$ 时，接收并对低电平脉冲开始计数，否则中断返回；当高电平脉冲宽 $< 3\text{ms}$ 时，接收并对低电平脉冲开始计数。

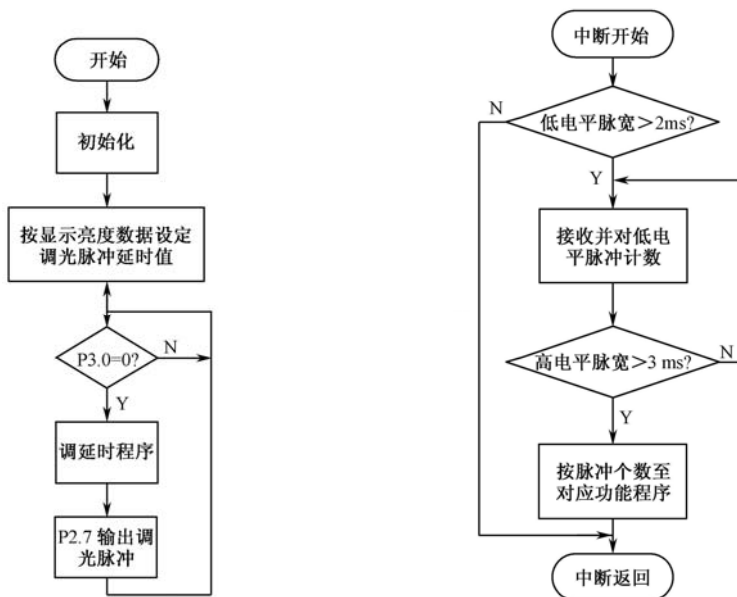


图 9-60 遥控码接收器程序的流程图

图 9-61 中断程序流程图

5. 按键扫描程序流程图

按键扫描程序流程如图 9-62 所示，扫描开始后，当有按键按下时，逐行扫描，否则返回。

6. 单片机遥控器部分程序代码

遥控部分程序代码说明如下：

```

//遥控发射器部分
#include<reg51.h>
#include<intrins.h>
#define k0 P0
#define k1 P1
#define uchar unsigned char
#define uint unsigned int
sbit out = P3^5;
//
uint a, b, m, n;
uchar key;
uchar code keyv[8] = {1, 2, 4, 8, 16, 32, 64, 128};
/*初始化函数*/
start()
{
    out = 0;
    IE = 0x00;
    IP = 0x01;
    TMOD = 0x22;
    TH1 = 0xf3;
    TL1 = 0xf3;
    EA = 1;
}
//1 ms 延时程序
delay( uint t )
{
    for(a = 0; b < t; a++)
        for(b = 0; b < 120; b++);
}
//按键功能函数
keyfunc()
{
    key = 0x00; k1 = 0xff;
    if(k0 != 0xff)
    {
        delay(20);
        if(k0 != 0xff)
        {
            while(k0 != 0xff);
            k1 = 0xfe;
            if(k0 != 0xff)

```

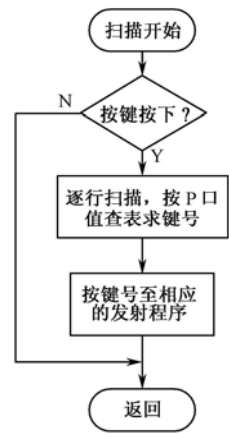


图 9-62 按键扫描程序流程图

```

{
    for( a = 0; a < 8; a++ )
    {
        if( ~k0 == keyv[a] )
        {
            key = a;
            test();
        }
    }
}
else
{
    k1 = 0xfd;
    if(k0 != 0xff)
    {
        for( a = 0; a < 8; a++ )
        {
            if( ~k0 == keyv[a] )
            {
                key = a + 8;
                test();
            }
        }
    }
}

{
    k1 = 0xfb;
    if(k0 != 0xff)
    {
        for( a = 0; a < 8; a++ )
        {
            if( ~k0 == keyv[a] )
            {
                key = a + 16;
                test();
            }
        }
    }
}
{

```

```

        k1 = 0xf7;
        if(k0 != 0xff)
        {
            for( a = 0; a < 8; a++ )
            {
                if(~k0 == keyv[a])
                {
                    key= a + 24;
                    test();
                }
            }
        }
    }
}

//发射函数
datasend()
{
    ET1= 1; TR1= 1; delay(3);
    ET1= 0; TR1 = 0; out = 0;    //40 kHz 发射时间 3 ms
    for( m = key; m > 0; m-- )
    {
        delay(1);                //延迟 1 ms
        ET1 = 1; TR1= 1; delay(1); ET1 = 0; TR1 = 0; out = 0;    //40 kHz 发 1 ms
    }
    delay(10);
}

//
//
test()
{

    switch(key)
    {
        case 0:key = key + 1; datasend(); break;
        case 1:key = key + 1; datasend(); break;
        case 2:key = key + 1; datasend(); break;
        case 3:key = key + 1; datasend(); break;
        case 4:key = key + 1; datasend(); break;
    }
}

```

```

        case 5:key = key + 1; datasend(); break;
        case 6:key = key + 1; datasend(); break;
        case 7:key = key + 1; datasend(); break;
        case 8:key = key + 1; datasend(); break;
        case 9:key = key + 1; datasend(); break;
        case 10:key = key + 1; datasend(); break;
        case 11:key = key + 1; datasend(); break;
        case 12:key = key + 1; datasend(); break;
        case 13:key = key + 1; datasend(); break;
        case 14:key = key + 1; datasend(); break;
        case 15:key = key + 1; datasend(); break;
        default : break;
    }
}
//主函数
main()
{
    start();
    while(1)
    {
        keyfunc();
    }
}
//40 kHz 发生器，定时中断 T1
void time_inter1(void) interrupt 3
{
    out = ~out;
}

```

7. 遥控接收部分的程序代码

遥控接收部分程序代码说明如下：

```

//遥控接收部分
#include<reg51.h>
#include<intrins.h>
#define uchar unsigned char
#define uint unsigned int
#define disout P1
sbit control = P3^1;
sbit sin = P3^0;
sbit A = P0^0;
sbit BB = P0^1;

```



```

sbit C = P0^2;
sbit D = P0^3;
sbit E = P0^4;
sbit F = P0^5;
sbit G = P0^6;
sbit H = P0^7;
sbit I = P2^0;
sbit J = P2^1;
sbit K = P2^2;
sbit L = P2^3;
sbit M = P2^4;
sbit N = P2^5;
sbit PP = P2^6;
sbit EN= P2^7;
//
uint m, n, q, z, i, j = 1;
uint key;
//
/*延时 1ms*/
delay(uint t)
{
    for(i = 0; i < t; i++)
        for(j = 0; j < 120; j++)
            ;
}
//初始化函数
start()
{
    EX0= 1;
    EA = 1;    //开总中断

}

//函数
loop()
{
    switch( disout & 0x07 )
    {
        case 0 : {z = 1; break;}
        case 1 : {z = 2; break;}
        case 2 : {z = 3; break;}
    }
}

```

```

        case 3 : { z = 4; break; }
        case 4 : { z = 5; break; }
        case 5 : { z = 6; break; }
        case 6 : { z = 7; break; }
        case 7 : { z = 8; break; }
        default : break;
    }
}
//主函数
main()
{
    start();
    loop();
    P2 = 0xfe;
    while(1)
    {
        while(sin == 1);

        delay(z);
        EN = 0;
        delay(1);
        EN = 1;
    }
}
//外部中断遥控接收数据
//外部中断 0
void inter0(void) interrupt 0
{
    EX0 = 0;
    key = 0;
    if(control == 0)
    {
        delay(1);
        if(control == 0)
        while(1)
        {
            while(control == 0);
            key++; key = 0;
            while(control == 1)
            {
                delay(1);

```

```

        q++;
        if( q > 2 )
        {
            goto second;
        }
    }
}

second:
switch(key)
{
    case 2 : { A = ~A; break;}
    case 3 : { BB = ~BB; break;}
    case 4 : { C = ~C; break;}
    case 5 : { D = ~D; break;}
    case 6 : { E = ~E; break;}
    case 7 : { F = ~F; break;}
    case 8 : { G = ~G; break;}
    case 9 : { H = ~H; break;}
    case 10 : { PP = ~PP; break;}
    case 11 : { N = ~N; break;}
    case 12 : { M = ~M; break;}
    case 13 : { L = ~L; break;}
    case 14 : { K = ~K; break;}
    case 15 : { J = ~J; break;}
    case 16 : { I = ~I; break;}
    case 17 : { if(disout==0x00){ disout=0xff;}
    else{ disout--; }loop();break;}
    default: break;
}

}

EX0=1;
}

```

附录A C51 库函数

1. CTYPE.H

CTYPE.H 文件包含对 ASCII 字符分类和字符转换的程序和原型，下面是程序的列表：

isalnum isprint toint
isalpha ispunct tolower
isctrl isspace _tolower
isdigit isupper toupper
isgraph isxdigit _toupper
islower toascii

字符转换和分类如下：

isalnum: 可重入是否是一个字母或数字字符；
isalpha: 可重入是否是一个字母字符；
isctrl: 可重入是否是一个控制字符；
isdigit: 可重入是否是一个十进制数；
isgraph: 可重入是否是一个除空格以外的可打印字符；
islower: 可重入是否是一个小写字母字符；
isprint: 可重入是否是一个可打印字符；
ispunct: 可重入是否是一个标点字符；
isspace: 可重入是否是一个空格；
isupper: 可重入是否是一个大写字母字符；
isxdigit: 可重入是否是一个十六进制数；
toascii: 可重入转换一个字符为一个 ASCII 码；
toint: 可重入转换一个十六进制数为一个十进制数；
tolower: 可重入测试一个字符如果是大写则转换成小写；
_tolower: 可重入无条件的转换一个字符为小写；
toupper: 可重入测试一个字符如果是大写则转换成小写；
_toupper: 可重入无条件的转换一个字符为大写。

字符转换和分类程序允许测试单个字符的各种属性并转换成不同的格式。

_tolower _toupper 和 **toascii** 程序作为宏来运行所有别的程序作为函数所有的宏定义和函数原型在包含文件 **CTYPE.H** 中

2. MATH.H

MATH.H 文件包含所有浮点数运算的程序的函数原型和定义别的数学函数也在这个文件中，所有的数学程序如下：

abs exp modf
acos fabs pow
asin floor sin
atan fmod sinh
atan2 fprestore sqrt
cabs fpsave tan
ceil labs tanh
cos log
cosh log10

数学程序如下所示：

acos/acos517：反余弦；
asin/asin517：反正弦；
atan/atan517：反正切；
atan2：分数的反正切；
ceil：取整；
cos/cos517：余弦；
cosh：双曲余弦；
exp/exp517：指数函数；
fabs：可重入取绝对值；
floor：小于等于指定数的最大整数；
fmod：浮点数余数；
log/log517：自然对数；
log10/log10517：常用对数；
modf：取出整数和小数部分；
pow：幂；
rand：随机数；
sin/sin517：正弦函数；
sinh：双曲正弦；
srand：初始化随机数发生器；
sqrt/sqrt517：平方根；
tan/tan517：正切函数；
tanh：双曲正切；
chkfloat：固有的，可重入的，检查 float 数的状态；
crol：固有的，可重入的，一个 unsigned char 向左循环位移；

cror: 固有的, 可重入的, 一个 unsigned char 向右循环位移;
irol: 固有的, 可重入的, 一个 unsigned int 向左循环位移;
iror: 固有的, 可重入的, 一个 unsigned int 向右循环位移;
lrol: 固有的, 可重入的, 一个 unsigned long 向左循环位移;
lror: 固有的, 可重入的, 一个 unsigned long 向右循环位移;
abs: 可重入取一个整数类型的绝对值;
atof/atof517: 转换一个字符串为一个 float;
atoi: 转换一个字符串为一个 int;
atol: 转换一个字符串为一个 long;
cabs: 可重入取一个字符类型的绝对值;
labs: 可重入取一个 long 类型的绝对值;
strtod/strtod 517: 一个字符串转换成一个 float;
strtol: 一个字符串转换成一个 long;
strtoul: 一个字符串转换成一个 unsigned long。

3. STDIO.H

STDIO.H 文件包含流 I/O 程序的函数原型和定义有:

getchar putchar sscanf

_getkey puts ungetchar

gets scanf vprintf

printf sprintf vsprintf

STDIO.H 包含文件也定义了 **EOF** 常数。

getchar: 可重入用 **_getkey** 和 **putchar** 程序读和显示一个字符;

_getkey: 用 8051A 串口读一个字符;

gets: 用 **getchar** 程序读和显示一个字符串;

printf/printf517: 用 **putchar** 程序写格式化数据;

putchar: 用 8051 串口写一个字符;

puts: 可重入用 **putchar** 程序写一个字符串和换行符\n;

scanf/scanf517: 用 **getchar** 程序读格式化数据;

sprintf/sprintf517: 写格式化数据到一个字符串;

sscanf/sscanf517: 从一个字符串读格式化数据;

ungetchar: 把一个字符放回到 **getchar** 输入缓冲区;

vprintf: 用 **putchar** 函数写格式化数据;

vsprintf: 写格式化数据到一个字符串。

4. STDLIB.H

STDLIB.H 文件包含下面类型转换和存储区分配程序的函数原型和定义。

atof init_mempool strtod

atoi malloc strtol

atol rand strtoul

calloc realloc

free srand

STDLIB.H 包含文件也定义了 **NULL** 常数。

存储区分配程序如下：

calloc：从存储池给一个数组分配存储区；

free：释放用 **calloc malloc** 或 **realloc** 分配彻底存储块；

init_mempool：初始化存储池的位置和大小；

malloc：从存储池分配一个块；

realloc：从存储池再分配一个块。

5. STRING.H

STRING.H 文件包含下面字符串和缓冲区操作程序的函数原型。

memccpy strchr strncpy

memchr strcmp strpbrk

memcmp strcpy strpos

memcpy strcspn strrchr

memmove strlen strpbrk

memset strncat strrpos

strcat strncmp strspn

STRING.H 包含文件也定义了 **NULL** 常数。

字符串操作程序如下：

strcat：连接两个字符串；

strchr：可重入返回一个字符串中指定字符第一次出现的位置指针；

strcmp：可重入比较两个字符串；

strcpy：可重入复制一个字符串到另一个；

strcspn：返回一个字符串中和第二个字符串的任何字符匹配的第一个字符的索引；

strlen：可重入字符串长度；

strncat：从一个字符串连接指定数目的字符到另一个字符串；

strncmp：比较两个字符串中指定数目的字符；

strncpy：从一个字符串复制指定数目的字符到另一个字符串；

strpbrk：返回一个字符串中和第二个字符串的任何字符匹配的最后一个字符的指针；

strpos：可重入返回一个指定字符在一个字符串中第一次出现的索引；

strrchr：可重入返回一个指定字符在一个字符串中最后出现的指针；

strrpbrk：返回一个字符串中和第二个字符串的任何字符匹配的最后一个字符的指针；

strrpos：可重入返回一个指定字符在一个字符串中最后出现的索引；

strspn：返回一个字符串中和第二个字符串中的任何字符不匹配的最后一个字符索引；

strstr：返回一个字符串中和另一个子字符串一样的指针。

附录B 语 法 信 息

B.1 致命错误信息

致命错误会立即终止编译，这些错误通常是命令行指定的无效选项的结果。当编译器不能访问一个特定的源包含文件时也会产生致命错误。

致命错误信息采用下面的格式：

C51 FATAL-ERROR –

ACTION <current action>
LINE: <line in which the error is detected>
ERROR: <corresponding error message>

C51 TERMINATED.

C51 FATAL-ERROR –

ACTION <current action>
FILE: <file in which the error is detected>
ERROR: <corresponding error message>

C51 TERMINATED.

下面说明 **Action** 和 **Error** 中可能的内容

1. Actions

ALLOCATING MEMORY: 编译器不能分配足够的存储区来编译指定的源文件。

CREATING LIST-FILE/OBJECT-FILE/WORKFILE: 编译器不能建立列表文件、OBJ 文件或工作文件，这个错误的出现可能是磁盘满或写保护，或文件已存在和只读。

GENERATING INTERMEDIATE CODE: 源文件包含的一个函数太大，不能被编译器编译成虚拟代码。尝试把函数分小或重新编译。

OPENING INPUT-FILE: 编译器不能发现或打开所选的源或包含文件。

PARSING INVOKE-/#PRAGMA-LINE: 当在命令行检测到参数计算，或在一个**#pragma** 中检测到参数计算，就产生这样的错误。

PARSING SOURCE-FILE/ANALYZING DECLARATIONS: 源文件包含太多的外部参考，减少源文件访问的外部变量和函数的数目。

WRITING TO FILE: 当写入列表文件，OBJ 文件或工作文件时遇到的错误。

2. Errors

‘(’ AFTER CONTROL EXPECTED: 一些控制参数需要用括号包含一个参数，当没有左括号时显示本信息。

‘)’ AFTER PARAMETER EXPECTED: 本信息表示包含没有参数的右括号。

BAD DIGIT IN NUMBER: 一个控制参数的数字参数包含无效字符，只能是十进制数。

CAN’T CREATE FILE: 在 **FILE** 行定义的文件名不能建立。

CAN’T HAVE GERERAL CONTROL IN INVOCATION LINE: 一般控制（例如，**EJECT**）不能包含在命令行，把这些控制用 **#pragma** 声明放在源文件中。

FILE DOES NOT EXIST: 没有发现定义在 **FILE** 行的文件。

FILE WRITE-ERROR: 因为磁盘空间不够，写到列表、预打印、工作或目标文件时出错。

IDENTIFIER EXPECTED: 当 **DEFINE** 控制没有参数时产生本信息。**DEFINE** 要求一个参数作为标识符。这和 C 语言的规则相同。

MEMORY SPACE EXHAUSTED: 编译器不能分配足够的存储区来编译指定的源文件。如果始终出现这个信息，应该把源文件分成两个或多个小文件再重新编译。

MORE THAN 100 ERRORS IN SOURCE-FILE: 在编译时检测到的错误超过 100 个，这使编译器终止。

MORE THAN 256 SEGMENTS/EXTERNALS: 在一个源文件中的参考超过 256 个。单个的源文件不能有超过 256 个函数或外部参考。这是 Intel 目标模块格式（OMF-51）的历史的限制。包含标量和/或 **bit** 声明的函数在 OBJ 文件中生成两个，有时候三个段定义。

NON-NULL ARGUMENT EXPECTED: 所选的控制参数需要用括号包含一个参数（例如一个文件名或一个数字）。

OUT OF RANGE NUMBER: 一个控制参数的数字参数超出范围。例如，**OPTIMIZE** 控制只允许数字 0~6，值 7 就将产生本错误信息。

PARSE STACK OVERFLOW: 解析堆栈溢出。如果源程序包含很复杂的表达式或如果块的嵌套深度超过 31 级，就会出现这个错误。

PREPROCESSOR LINE TOO LONG 32K: 一个中间扩展长度超过 32 KB 字符。

PREPROCESSOR MACROS TOO NESTED: 在宏扩展期间，预处理器所用的堆栈太大。这个信息通常表示一个递归的宏定义，但也可表示一个宏嵌套太多。

RESPECIFIED OR CONFLICTING CONTROL: 一个命令行参数指定了两次，或命令行参数冲突。

SOURCE MUST COME FROM A DISK-FILE: 源和包含文件必须存在。控制台 **CON** : , : **CI**: , 或类似的设备不能作为输入文件。

UNKNOWN CONTROL: 所选的控制参数不认识。

B.2 语法和语义错误信息

语法和语义错误一般出现在源程序中，它们确定实际的编程错误，当遇到这些错误

时，编译器尝试绕过错误继续处理源文件；当遇到更多的错误时，编译器输出另外的错误信息。但是，不产生 OBJ 文件。

语法和语义错误在列表文件中生成一条信息。这些错误信息用下面的格式：

*** **ERROR** *number* **IN LINE** *line* **OF file:***error message*

这里：

- number** 为错误号；
- line** 为对应源文件或包含文件的行号；
- file** 为产生错误的源或包含文件名；
- error message** 为对错误的叙述说明。

下表按错误号列出了语法和语义错误，错误信息列出了主要说明和可能的原因和改正。

序号	错误信息和说明
100	跳过不可打印字符 0x?? 在源文件中发现一个非法字符（注意不检查注释中的字符）
101	字符串没结束，一个字符串没有用双引号 “ ” 终止。
102	字符串太长 一个字符串不能超过 4096 个字符。用串联符号（ ‘\’ ）在逻辑上可延长字符串超过 4096 个字符。这个模式的行终止符在词汇分析时是连续的
103	无效的字符常数 一个字符常数的格式无效。符号 ‘\c’ 是无效的，除非 c 是任何可打印的 ASCII 字符
125	声明符太复杂（20） 一个目标的声明可包含最多 20 个类型修饰符（ ‘[’ ， ‘]’ ， ‘*’ ， ‘(’ ， ‘)’ ’ ），这个错误经常伴随着错误 126
126	类型堆栈下溢 类型声明堆栈下溢，这个错误通常是错误 125 的副产品
127	无效存储类 一个目标用一个无效的存储空间标识符声明。如果一个目标在一个函数外用存储类 auto 或 register 声明，就会产生本错误
129	在 ‘标记’ 前缺少 ‘;’ 本错误通常表示前一行缺少分号。当出现本错误时，编译器会产生很多错误信息
130	值超出范围 在一个 using 或 interrupt 标识符后的数字参数是无效的。using 标识符要求一个 0~3 之间的寄存器组号。interrupt 标识符要求一个 0~31 之间的中断矢量号
131	函数参数重复 一个函数有相同的参数名，在函数声明中参数名必须是唯一的
132	没在正式的参数列表 一个函数的参数声明用了一个名称没在参数名列表中。例如 char function(v0,v1,v2) char *v0,*v1,*v5; /* ‘v5’ 没在正式列表中 */ { /* ... */ }

序号	错误信息和说明
134	函数的 xdata/idata/pdata/data 不允许 函数通常位于 code 存储区，不能在别的存储区运行。函数默认定义为存储类型 code
135	bit 的存储类错 bit 标量的声明可能包含一个 static 或 extern 存储类。register 或 alien 类是无效的
136	变量用了 ‘void’ void 类型只允许作为一个不存在的返回值，或一个函数的空参数列表 [void func(void)]，或和一个指针组合 (void *)
138	Interrupt()不能接收或返回值 一个中断函数被定义了一个或多个正式的参数，或一个返回值。中断函数不能包含调用参数或返回值
140	位在非法的存储空间 bit 标量的定义可以包含可选的存储类型 data。如果没有存储类型则默认为 data，因为位通常在内部数据存储区。当试图对一个 bit 标量定义别的数据类型时会产生本错误
141	临近标志语法错误：期待别的标志，... 编译器所见的标志是错误的，参考所显示的期待的内容
142	无效的基地址 一个 SFR 或 SBIT 声明的基地址是错误的，有效的基地址范围在 0x80~0xff 之间，如果用符号基地址^位号声明，则基地址必须是 8 的倍数。
143	无效的绝对位地址 sbit 声明中的绝对位地址必须在 0x80~0xff 之间
144	基地址^位号：无效的位号 sbit 声明中定义的位号必须在 0~7 之间
145	未知的 SFR
146	
146	无效 SFR 一个绝对位（基地址^位号）的声明包含一个无效的基地址标识符。基地址必须是已经声明的 sfr。任何别的名称是无效的
147	目标文件太大 单个目标文件不能超过 65 535 (64 KB-1)
149	struct/union 包含函数成员 struct 或 union 不能包含一个函数类型的成员，但是，指向函数的指针是可以的
150	struct/union 包含一个 bit 成员 一个 union 不能包含 bit 类型成员，这是 8051 的结构决定的
151	struct/union 自我关联 一个结构不能包含自己
152	位号超出位域 位域声明中指定的位号超过给定基类的位号
153	命名的位域不能为零 命名的位域为零只要未命名的位域允许为零
154	位域指针 指向位域的指针不允许
155	位域要求 char/int 位域的基类要求 char 或 int，unsigned char 和 unsigned int 类型也行
156	alien 只允许对函数

序号	错误信息和说明
157	alien 函数带可变参数 存储类 alien 只对外部 PL/M-51 函数允许。符号 (char *,...) 在 alien 函数中是非法的。PL/M-51 函数通常要求一个固定的参数表
158	函数包含未命名的参数 一个函数的参数列表定义包含一个未命名的抽象类型定义, 这个符号只允许在函数原型中
159	void 后面带类型 函数的原型声明可包含一个空参数列表 (例如, int func(void)), 在 void 后不能再有类型定义
160	void 无效 void 类型只在和指针组合时, 或作为一个函数的不存在的返回值时才是合法的
161	忽视了正式参数 在一个函数内, 一个外部函数的声明用了一个没有类型标识符的参数名列表 (例如, extern yylex(a,b,c);)
180	不能指向一个函数 指向一个函数的类型是无效的, 尝试用指针指向一个函数
181	操作数不兼容 对给定的操作符, 至少一个操作数类型是无效的 (例如, ~float_type)
183	左值不能修改 要修改的目标位于 code 存储区或有 const 属性, 因此不能修改
184	sizeof: 非法操作数 sizeof 操作符不能确定一个函数或位域的大小
185	不同的存储空间 一个目标声明的存储空间和前一个同样目标声明的存储空间不同
186	解除参照无效 一个内部编译器问题会产生本信息, 如果本错误重复出现请和技术支持接洽
187	不是一个左值 所需的参数必须是一个可修改的目标地址
188	未知目标大小 因为没有数组的维数, 或间接通过一个 void 指针, 一个目标的大小不能计算
189	‘&’ 对 bit/sfr 非法 取地址符‘&’: 不允许对 bit 目标或特殊函数寄存器 (sfr)
190	& 不是一个左值 尝试建立一个指针指向一个未知目标
193	错误操作数类型 当对一个给定的操作符用了非法的操作数类型时产生本错误, 例如, 无效的表达式: 如 bit*bit, ptr+ptr 或 ptr*anything, 这个错误信息包括引起错误的操作符 下面的操作对 bit 类型的操作数是可行的: ● 赋值 (=) ● OR/复合 OR (, =) ● AND/复合 (AND & &=) ● =XOR/复合 XOR (^ ^=) ● bit 比较 (== !=) ● 取反 (~) ● bit 操作数可和别的数据类型在表达式中混用, 在这种情况下类型转换将自动执行
194 *	间接指向一个未知大小的目标 间接操作符*不能和 void 指针合用, 因为指针所指的目标的大小是未知的

序号	错误信息和说明
195	间接非法 *操作符不能用到非指针参数
196	存储空间可能无效 转换一个常数到一个指针常数产生一个无效的存储空间。例如 <code>char*p=0x91234</code>
198	<code>sizeof</code> 返回零 <code>sizeof</code> 操作符返回零
199	<code>'->'</code> 的左边要求 <code>struct/union</code> 指针 <code>-></code> 操作符的左边参数必须是一个 <code>struct</code> 指针或一个 <code>union</code> 指针
200	<code>'.'</code> 左边要求 <code>struct/union</code> <code>'.'</code> 操作符的左边参数要求必须是 <code>struct</code> 或 <code>union</code> 类型
201	未定义的 <code>struct/union</code> 给定的 <code>struct</code> 或 <code>union</code> 名是未知的
202	未定义的标识符 给定的标识符是未定义的
203	错误的存储类（参考名） 本错误表示编译器的一个问题，如果重复出现请接洽技术支持
204	未定义的成员 给定的一个 <code>struct</code> 或 <code>union</code> 成员名是未定义的
205	不能调用一个中断函数 一个中断函数不能像一个正常函数一样调用，中断的入口和退出代码是特殊的
207	参数列表声明为 <code>'void'</code> 参数列表声明为 <code>void</code> 的函数不能从调用者接收参数
208	太多的实参 函数调用包含太多的实参
209	太少的实参 调用函数包含太少的实参
210	太多的嵌套调用 函数的嵌套调用不能超过 10 级
211	调用不是对一个函数 一个函数的调用项不是对一个函数或函数指针求值
212	间接调用寄存器的参数不匹配 通过一个指针的间接函数调用不包含实际的参数。一个例外是当所有的参数可以通过寄存器传递，这是由于 Cx51 所用的传递参数的方法决定的。被调用的函数名必须是已知的，因为参数写到被调用函数的数据段。但是，对间接调用来说，被调用函数的名称是未知的
213	赋值符的左边不是一个左值 赋值符的左边要求一个可修改目标的地址
214	非法指针转换 <code>bit</code> , <code>float</code> 或集合类型的目标不能转换为指针
215	非法类型转换 <code>struct/union/void</code> 不能转换为任何别的类型
216	标号用在非数组中，或维数超出 一个数组引用包含太大的维数，或目标不是一个数组
217	非整数索引 一个数组的维数表达式必须是 <code>char</code> , <code>unsigned char</code> , <code>int</code> 或 <code>unsigned int</code> 类型，别的类型都是非法的

序号	错误信息和说明
218	控制表达式用了 void 类型 在一个 while、for 或 do 的限制表达式中不能用类型 void
219	Long 常数缩减为 int 一个常数表达式的值必须能用一个 int 类型表示
220	非法常数表达式 期望一个常数表达式。目标名、变量或函数不允许出现在常数表达式中
221	非常数 case/dim 表达式 一个 case 或一个维数 ([]) 必须是一个常数表达式
222	被零除
223	被零取模 编译器检测到一个被零除或取模
225	表达式太复杂, 需简化 一个表达式太复杂, 必须分成两个或多个子表达式
226	重复的 struct/union/enum 标记 一个 struct、union 或 enum 名早已定义
227	表示一个 union 标记 一个 union 名称早已定义为别的类型
228	表示一个 struct 标记 一个 struct 名早已定义为别的类型
229	表示一个 enum 标记 一个 enum 名早已定义为别的类型
230	未知的 struct/union/enum 标记 指定的 struct、union 或 enum 名未定义
231	重复定义 指定的名称已被定义
232	重复标号 指定的标号已定义
233	未定义标号 表示一个标号未定义, 有时候这个信息会在实际的标号的几行后出现。这是所用的未定义标号的搜索方法引起的
234	‘{’, 堆栈范围溢出 (31) 超过了最多 31 个嵌套块, 超出的嵌套块被忽略
235	参数<数字>: 不同类型 函数声明的参数类型和函数原型中的不同
236	参数列表的长度不同 函数声明中的参数数目和函数原型中的不同
237	函数早已定义 试图声明一个函数体两次
238	重复成员
239	重复参数 试图定义一个已存在的 struct 成员或函数参数
240	超出 128 个局部 bit 在一个函数内不能超过 128 个 bit 标量

序号	错误信息和说明
241	<p>auto 段太大</p> <p>局部目标所需的空間超过模式的极限。最大的段大小定义如下：</p> <p>SMALL 128 B</p> <p>COMPACT 256 B</p> <p>LARGE 65 535 B</p>
242	<p>太多的初始化软件</p> <p>初始化软件的数目超过初始化目标的数量</p>
243	<p>字符串超出范围</p> <p>字符串中的字符数目超出字符串初始化的数目</p>
244	<p>不能初始化，错误的类型或类</p> <p>试图初始化一个 bit 或 sfr</p>
245	<p>未知的 pragma，跳过本行</p> <p>#pragma 状态未知，所以整行被忽略</p>
246	<p>浮点错误</p> <p>当一个浮点参数超出 32 位的范围就产生本错误。32 位 IEEE 值的范围是： $\pm 1.175494\text{E}-38 \sim \pm 3.402823\text{E}+38$。</p>
247	<p>非地址/常数初始化</p> <p>一个有效的初始化表达式必须是一个常数值求值或一个目标名加或减去一个常数</p>
248	<p>集合初始化需要大括号</p> <p>给定 struct 或 union 初始化缺少大括号 ({})</p>
249	<p>段<名>：段太大</p> <p>编译器检测到一个数据段太大。一个数据段的最大的大小由存储空间决定</p>
250	<p>‘\esc’；值超过 255</p> <p>一个字符串常数中的转义序列超过有效值范围，最大值是 255</p>
252	<p>非法八进制数</p> <p>指定的字符不是一个有效的八进制数</p>
252	<p>主要控制放错地方，行被忽略</p> <p>主要控制必须被指定在 C 模块的开头，在任何#include 命令或声明前</p>
253	<p>内部错误 ASMGEN\CLASS</p> <p>在下列情况下出现本错误：</p> <ul style="list-style-type: none"> • 一个内在函数（例如_testbit_）被错误激活，这种情况是在没有函数原型存在和实参数目或类型错误，对这种原因，必须使用合适的声明文件（INTRINS.H， STRING.H）； • Cx51 确认一个内部一致性问题
255	<p>switch 表达式有非法类型</p> <p>在一个 switch 表达式没有合法的数据类型</p>
256	<p>存储模式冲突</p> <p>一个包含 alien 属性的函数只能包含模式标识符 small。函数的参数必须位于内部数据区。这适用于所有的外部 alien 声明和 alien 函数。例如：</p> <pre>alien plm_func(char c) large { ... }</pre> <p>产生错误 256</p>
257	<p>alien 函数不能重入</p> <p>一个包含 alien 属性的函数不能同时包含 reentrant 属性，函数参数不能跳过虚拟堆栈传递，这适用于所有的外部 alien 声明和 alien 函数</p>

序号	错误信息和说明
258	<p>struct/union 成员的存储空间非法</p> <p>非法空间的参数被忽略</p> <p>一个结构的成员或参数不能包含一个存储类型标识符。但指针所指的目标可能包含一个存储类型。例如：</p> <pre>struct vp{char code c;int xdata i;};</pre> <p>产生错误 258</p> <pre>struct v1{char c;int xdata *i;};</pre> <p>是正确的 struct 声明</p>
259	<p>指针不同的存储空间</p> <p>一个空指针被关联到别的不同存储空间的空指针。例如：</p> <pre>char xdata *p1;</pre> <pre>char idata *p2;</pre> <pre>p1 = p2; //不同的存储空间</pre>
259	<p>指针断开</p> <p>一个空指针被关联到一些常数值，这些值超过了指针存储空间的值范围，例如：</p> <pre>char idata *p1 = 0x1234; //结果是 0x34</pre>
261	<p>reentrant()内有 bit</p> <p>一个可重入属性的函数的声明中不能包含 bit 目标。例如：</p> <pre>int func1(int i1) reentrant {</pre> <pre>bit b1,b2; //不允许</pre> <pre>return(i1-1);</pre> <pre>}</pre>
262	<p>‘using/disable’：不能返回 bit 值</p> <p>用 using 属性声明的函数和禁止中断（#pragma disable）的函数不能返回一个 bit 值给调用者，例如：</p> <pre>bit test(void) using 3</pre> <pre>{</pre> <pre>bit b0;</pre> <pre>return(b0);</pre> <pre>}</pre> <p>产生错误 262</p>
263	<p>保存/恢复：堆栈保存溢出/下溢</p> <p>#pragma save 的最大嵌套深度是 8 级。堆栈的 pragma save 和 restore 工作根据 LIFO 后进先出规则</p>
264	<p>内在的‘<内在的名称>’声明/激活错误</p> <p>本错误表示一个内在的函数错误定义（参数数目或省略号）。如果用标准的.H 文件就不会产生本错误。确认使用了 Cx51 所有的.H 文件，不要尝试对内在的库函数定义自己的原型</p>
265	<p>对非重入函数递归调用</p> <p>非重入函数不能被递归调用，因为这样会覆盖函数的参数和局部数据。如果需要递归调用，需声明函数为可重入函数</p>
267	<p>函数定义需要 ANSI 类型的原型</p> <p>一个函数被带参数调用，但是声明是一个空的参数列表。函数原型必须有完整的参数类型，这样编译器就可能通过寄存器传递参数，和适合应用的调用机制</p>
268	<p>任务定义错误（任务 ID/优先级/using）</p> <p>任务声明错误</p>

序号	错误信息和说明
271	<p>‘asm/endasm ’ 控制放错地方</p> <p>asm 和 endasm 声明不能嵌套。endasm 要求一个汇编块，前面用 asm 开头。例如：</p> <pre>#pragma asm ... 汇编指令 ... #pragma endasm</pre>
272	<p>‘asm’ 要求激活 SRC 控制</p> <p>在一个源文件中使用 asm 和 endasm，要求文件用 SRC 控制编译，编译器就会生成汇编源文件，然后可以用 A51 汇编</p>
273	<p>‘asm/endasm’ 在包含文件中不允许</p> <p>在包含文件中不允许 asm 和 endasm，为了调试在包含文件不能有任何的可执行代码</p>
274	<p>非法的绝对标识符</p> <p>绝对地址标识符对位目标、函数和局部函数不允许。地址必须和目标的存储空间一致。例如，下面的声明是无效的，因为间接寻址的范围是 0x00~0xff，如</p> <pre>idata int _at_ 0x1000;</pre>
278	<p>常数太大</p> <p>当浮点参数超出 32 位的浮点值范围就产生本错误。32 位 IEEE 值的范围是：</p> $\pm 1.175494\text{E}-38 \sim \pm 3.402823\text{E}+38$
279	<p>多次初始化</p> <p>试图多次初始化一个目标</p>
280	<p>没有使用符号/标号/参数</p> <p>在一个函数中声明了一个符号、标号或参数，但没有使用</p>
281	<p>非指针类型转换为指针</p> <p>引用的程序目标不能转换成一个指针</p>
282	<p>不是一个 SFR 引用</p> <p>本函数调用要求一个 SFR 作为参数</p>
283	<p>asmparms 参数不适合寄存器</p> <p>参数不适合可用的 CPU 寄存器</p>
284	<p><名称>：在可覆盖空间，函数不再可重入</p> <p>一个可重入函数包含对局部变量的明确的存储类型标识符，函数不再完全可重入</p>
300	<p>注释未结束</p> <p>一个注释没有一个结束符（*/）</p>
301	<p>期望标识符</p> <p>一个预处理器命令期望一个标识符</p>
302	<p>误用#操作符</p> <p>字符操作符 # 没有带一个标识符</p>
303	<p>期望正式参数</p> <p>字符操作符# 没有带一个标识符表示当前所定义的宏的一个正式参数名</p>
304	<p>错误的宏参数列表</p> <p>宏参数列表没有一个大括号或逗号分开的标识符列表</p>
305	<p>string/char 常数未结束</p> <p>一个字符串字符常数是无效的，典型的是后引号丢失</p>

序号	错误信息和说明
306	宏调用未结束 预处理器在收集和扩展一个宏调用的实际的参数时遇到输入文件的结尾
307	宏名称：参数计算不匹配 在一个宏调用中实际的参数数目和宏定义的参数数目不匹配，本错误表示指定了太少的参数
308	无效的整数常数表达式 一个 <code>if/elif</code> 命令的数学表达式包含一个语法错误
309	错误或缺少文件名 在一个 <code>include</code> 命令中的文件名参数是无效的或不存在的
310	条件嵌套过多于 20 源文件包含太多的条件编译嵌套命令，最多允许 20 级嵌套
311	<code>elif/else</code> 控制放错地方
312	<code>endif</code> 控制放错地方 命令 <code>elif</code> 、 <code>else</code> 和 <code>endif</code> 只有在 <code>if</code> 、 <code>ifdef</code> 和 <code>ifndef</code> 命令中是合法的
313	不能清除预定义的宏名称 试图清除一个预定义宏。用户定义的宏可以用 <code>#undef</code> 命令删除，预定义的宏不能清除
314	#命令语法错误 在一个预处理器命令中，字符 <code>#</code> 必须跟一个新行或一个预处理器命令名（例如， <code>if/define/ifdef ...</code> ）
315	未知的#命令名称 预处理器命令是未知的
316	条件未结束 到文件结尾 <code>endif</code> 的数目和 <code>if</code> 或 <code>ifdef</code> 的数目不匹配
318	不能打开文件文件名 指定的文件不能打开
319	文件不是一个磁盘文件 指定的文件不是一个磁盘文件，文件不能编辑
320	用户自定义的内容 本错误号未预处理器的 <code>#error</code> 命令保留， <code>#error</code> 命令产生错误号 320，送出用户定义的错误内容，终止编译器生成代码
321	缺少<字符> 在一个 <code>include</code> 命令的文件名参数中，缺少结束符，例如， <code>#include<stdio.h</code>
325	正参名称重复 一个宏的正参只能定义一次
326	宏体不能以 <code>##</code> 开始或结束 <code>##</code> 不能是一个宏体的开始或结束
327	宏名：超过 50 个参数 每个宏的参数数目不能超过 50

参 考 文 献

- [1] 戴佳, 戴卫恒. 51 单片机 C 语言应用程序设计实例精讲. 北京: 电子工业出版社, 2006.
- [2] 刘文涛. 单片机语言 C51 典型应用设计. 北京: 人民邮电出版社, 2005.
- [3] 边春元. C51 单片机典型模块设计与应用. 北京: 机械工业出版社, 2008.
- [4] 秦志强. C51 单片机应用与 C 语言程序设计. 北京: 电子工业出版社, 2006.
- [5] 陈涛. 单片机应用及 C51 程序设计北京: 机械工业出版社, 2008.
- [6] 张道德. 单片机接口技术(C51 版). 北京: 中国水利水电出版社, 2007.
- [7] 祁伟, 杨亨. 单片机 C51 程序设计教程与实验. 北京: 北京航空航天大学出版社, 2006.
- [8] 于永. 51 单片机 C 语言常用模块与综合系统设计实例精讲. 北京: 电子工业出版社, 2007.